LATENT LATTICE LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Standard autoregressive transformers route all per-token computation through a single active latent. This latent is responsible for querying and processing the "memory" from the past steps, forming new memory (updating the KV cache) for the future steps, producing residual updates to its own state, and finally decoding into the next token. The memory is a *write-once, read-only* log of the latent's past states, while all active computation still rests on that one latent vector. Recent work increases the computation performed by this single latent either along depth – by adding more layers or looping – or along time, by letting it persist across steps rather than being decoded at each one. However, the model remains bottlenecked, relying on a *single latent* to perform every role. We instead ask: how can we *increase the computational latent capacity* available to a language model, and what does that buy us? To that end, we introduce an *additional horizontal latent* at each Lattice layer, updated continuously along the temporal dimension. Together with the standard vertical latent that runs along the network's depth axis, these horizontal recurrent states form our **Lattice of Latents**. Unlike hybrid Transformer–SSM architectures, where recurrent state typically remains internal to a sequence-mixing operator (primarily focused on replacing softmax-attention and KV cache), our horizontal latent acts as an *explicit companion computational state* to the vertical latent, jointly driving attention, KV-cache writes, MLP computation, and decoding. We propose several mechanisms to increase the expressivity of these horizontal latents while retaining token-parallel teacher-forced computation within each pass. Across pretraining budgets of 10B–30B tokens, Lattice models consistently improve language-modeling quality and downstream transfer. After pretraining on 20B FineWeb-Edu tokens, our 406M-parameter Latent Lattice Language Model lowers perplexity from 15.41 to 14.40 relative to a parameter and compute-matched Transformer, while also matching a 609M-parameter, 40-layer Transformer that uses $1.5\times$ as many parameters, $1.8\times$ the KV-cache footprint, and $1.5\times$ as many inference FLOPs. The gains also extend to downstream evaluation and mathematical reasoning after finetuning. **Project Page:** `https://latent-lattice-language-model.netlify.app/`

1 INTRODUCTION

Language models accept raw text as input, transform it through internal representations, and predict subsequent text. Like most learned systems, this process relies on some form of latent computation. But what, precisely, is the latent computation in a large language model?

In standard autoregressive LLMs, at each token timestep, an input token is encoded into a latent representation and processed through a depth-N stack of nonlinear layers. At every layer, this latent produces query, key, and value tensors: the query reads from a running store, while the key and value are written into it for later retrieval. The MLP further transforms the latent, and the result is passed to the next layer, or, at the final layer, decoded into the next token. Thus, a single running latent is responsible for both reading from and writing to memory, generating residual updates to itself, and producing the eventual output. The memory it writes to, the KV cache, is simply a persistent, read-only record of the latent's past states – an append-only log, while the *latent remains the sole active computation*. Recent work increases this active computation along two axes. One line increases depth: adding more layers or looping over the same layers multiple times Dehghani et al. (2019); Geiping et al. (2025); Zhu et al. (2026) for a single text token before it is decoded. A separate

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.

line of work Hao et al. (2024); Wu et al. (2026); Zeng et al. (2026); Shen et al. (2025) increases computation along the temporal dimension by adding latent computation steps between text tokens so that the latent can undergo more computation before being decoded into the next text token.

In both cases, the *same single latent, alone, performs all jobs*. Since a single latent must shoulder all of this, we ask whether language models can benefit from increasing the underlying computational capacity available to their latents. Rather than naively increasing the dimensionality of this one running latent, we propose a *lattice of computational latents*: each Lattice layer is given its own latent vector (r_t^ℓ, denoted by orange arrows in Fig. 1), updated primarily across timesteps rather than across depth, alongside the latent that already runs through the network's depth (h_t^ℓ, denoted by blue arrows in Fig. 1). This new latent acts as a companion to the vertical latent, directly assisting it in all of its tasks. We call this architecture the **Lattice of Latents**, and a language model built on it the **Latent Lattice Language Model (LLLM)**.

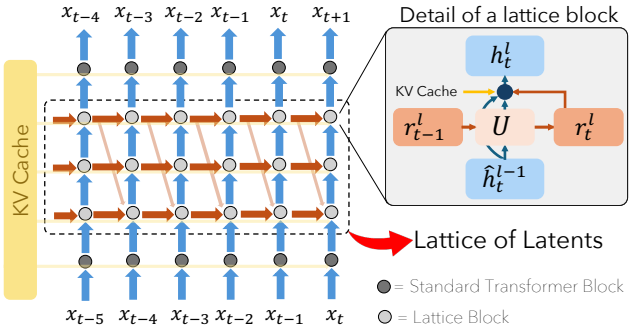


Figure 1: **Latent Lattice Language Model.** See webpage for animations, and comparison with base transformer.

Being more precise, how is this horizontal flow of information leveraged? At each step of computation – layer ℓ, timestep t – two streams of active information are available: the vertical latent $h_t^{\ell-1}$ from the layer below, and the horizontal latent r_t^ℓ, formed by updating the same layer's previous state r_{t-1}^ℓ. Both latents jointly drive that layer's attention and MLP, and, at the final layer, its output decoding. The horizontal latent assists the vertical latent throughout: conditioning attention, modulating the MLP and the output decoding. A central question in our work is how we can update the horizontal latent such that (a) it is expressive, i.e., contributes to improved pretraining metrics, and (b) preserves the standard transformer's parallel, teacher-forced training.

Towards these goals, we start by defining an affine Mamba-style (Gu & Dao, 2024) update for horizontal latents, in which the horizontal latent r_t^ℓ is updated from r_{t-1}^ℓ using the incoming vertical latent $h_t^{\ell-1}$. This affine associative-scan-based update ensures that training is completely parallelizable. While this recurrence is closely related to those used in Mamba and hybrid Transformer–SSM models, *it serves a different purpose:* their recurrent state typically remains internal to a token-mixing or memory operator, primarily focused on replacing softmax attention and the KV cache at certain layers. In contrast, ours remains an *explicit companion computational state that conditions or steers everything the vertical latent processes* – softmax attention, its KV-cache writes, the MLP, and decoding. However, such a Mamba-style affine update formulation using the input vertical latent state, ties the horizontal latent's own information capacity to that of the vertical latent. To address this, we further study multiple enhancements: (a) additionally passing the input token embedding directly into each layer's horizontal update; (b) while keeping each horizontal latent attached to one particular layer or abstraction, enriching the readout of r_t^ℓ — not r_t^ℓ itself — with information from $r_t^{\ell-1}$ before it steers attention, MLP, or output; and (c) connecting the previous timestep's horizontal latent at the Lattice's top layer, r_{t-1}^{top}, to the update of the Lattice's bottom layer horizontal latent, r_t^{bottom}, at the current timestep. The first two enhance information density and per-timestep layer-wise nonlinearity, respectively, while preserving training parallelism. The third, although enhancing cross-timestep nonlinearity, breaks single-pass teacher-forced parallelism. To recover token-parallelism, inspired by (Wu et al., 2026; Zeng et al., 2026), we leverage a small number of Jacobi iterations – a mechanism that preserves token parallelism within each pass, but performs multiple passes over the Lattice.

Across pretraining budgets from 10B to 30B FineWeb-Edu tokens, the Lattice of Latents improves language-modeling quality at matched parameters and compute (Tab. 1, Tab. 6, Fig. 4). At 20B tokens, a 406M-parameter LLLM lowers validation perplexity from 15.41 to 14.40 relative to a size- and compute-matched Transformer – thereby *matching* a $1.5\times$ larger, 40-layer Transformer that also uses $1.8\times$ the KV cache and $1.5\times$ the inference FLOPs – and with three Jacobi sweeps surpasses

Approach	Params (M)	GFLOP/token		FineWeb Edu	Pile	LAMBADA		Nemo CC
		Train	Infer			OAI	Stand.	
400M-parameter models								
Base-24L (width $d_v = 1024$)	406.7	0.71	0.71	15.410	26.276	41.172	73.648	14.768
LLLM $(945, 256)$	406.5	0.72	0.72	14.545	25.031	38.835	65.780	14.278
LLLM $(908, 256)$ inj+vs	405.6	0.72	0.72	14.400	**24.100**	38.513	63.259	14.116
LLLM-410 J3 $(945, 256)$	406.9	1.85	1.85	**14.026**	24.107	**37.531**	**62.923**	**13.694**
↪ *sequential rollout at inference (no Jacobi sweeps)*	406.9	1.85	0.72	14.733	25.461	38.737	68.398	—
500M-parameter models								
LLLM $(512, 512)$, 40L	495.2	0.94	0.94	14.368	24.972	38.119	64.649	14.127
600M-parameter models								
Base-24L (width $d_v = 1620$)	603.3	1.04	1.04	14.690	24.443	38.666	68.939	14.018
Base-24L (width $d_v = 1472$)	606.2	1.06	1.06	14.649	24.610	39.272	67.624	14.221
Base-40L (width $d_v = 1024$)	609.1	1.11	1.11	14.400	24.234	38.624	67.508	13.883
LLLM $(1024, 1024)$	603.0	1.10	1.10	**13.704**	**24.200**	**37.012**	**60.132**	**13.443**

Table 1: **Language-modeling Perplexity (↓) after 20B-token pretraining.** Fineweb-Edu are in-distribution results, while others are OOD. 400M LLLM (908, 256) `inj+vs` (input-injection + vstream) performs better than 600M base transformers. Similar perf. at 30B training scale Tab. 6.

it outright, at higher inference compute. These gains persist out of distribution to Pile (Gao et al., 2020), LAMBADA (Paperno et al., 2016), and Nemotron-CC (Su et al., 2025), and transfer to fine-tuned mathematical reasoning. We further show benefits of incorporating the Lattice of Latents into complementary temporal-computation settings—Transformers with pause tokens and PonderLM-2-style latent reasoning Zeng et al. (2026). Finally, our ablations (Fig. 4) isolate the contribution of each horizontal-latent mechanism and show that, at a fixed budget, dividing capacity between the vertical and horizontal latents beats spending it on width or depth alone.

2 Related Work

Recent work increases a language model's per-token computation along two axes. Along *depth*, models add layers or repeatedly apply a shared stack (Dehghani et al., 2019; Geiping et al., 2025; Zhu et al., 2026); along *time*, continuous-latent methods carry a single evolving hidden state through additional latent steps before emitting the next text token (Hao et al., 2024; Shen et al., 2025; Zeng et al., 2026; Wu et al., 2026). Both directions primarily increase the number of updates applied to one active computational stream, and often couple additional computation with additional persistent KV memory—through extra layers, loop-specific caches, or latent-thought positions. LLLM instead distributes active computation across persistent layer-wise latents, increasing computational latent capacity without increasing the KV-cache size.

Our affine horizontal update belongs to the broader family of scan-parallel recurrent models, including Mamba (Gu & Dao, 2024; Dao & Gu, 2024), Gated DeltaNet (Yang et al., 2025), and DeltaProduct (Siems et al., 2025). Our Jacobi refiner is inspired by methods that parallelize latent-thought recurrence through iterative refinement (Zeng et al., 2026; Wu et al., 2026); complementary work ParaRNN parallelizes general nonlinear recurrences via Newton iterations (Danieli et al., 2025).

Compared to hybrid Mamba–Transformer models, LLLM differs in architectural role. Such models use recurrent state internally within an SSM sequence-mixing branch—either in place of attention at selected layers (Lieber et al., 2024) or alongside attention heads (Dong et al., 2024). In contrast, our horizontal latent remains an explicit companion computational state to the vertical latent stream: LLLM retains full softmax attention and uses the horizontal latent to condition everything vertical stream processes – attention and its KV-cache writes, modulate the MLP, and contribute to decoding.

Several concurrent works study related recurrent or multi-stream Transformer architectures (Wang et al., 2026; Huang et al., 2026; Monea et al., 2026; Mozer et al., 2026; Cai et al., 2026). Appendix A.3 provides a more detailed comparison.

3 METHOD

We propose the **Lattice of Latents**, a transformer whose computation is carried not by a single latent vector per token but by a lattice of latents spanning two axes. Along the *vertical* axis (the depth of the network), the single per-token latent flows bottom-to-top through the layers exactly as in a standard transformer — recomputed fresh at every timestep, updated at each layer by attention (which reads from and writes to memory) and by a position-wise MLP transformation, and finally decoded to the output. Along a new *horizontal* axis (the time dimension), the Lattice of Latents adds one latent vector at each layer of an intermediate Lattice block, each running across timesteps rather than through depth. Every such horizontal latent stays within its own layer, at its own abstraction level, and assists the vertical latent in attention and MLP computation; the topmost horizontal latent additionally contributes to output decoding. Thus, the proposed system offers a *larger computational latent capacity* by maintaining an active lattice of latents. We begin by defining the per-token computational graph of the vertical latent, followed by the full formulation of our horizontal latent.

3.1 COMPUTATION WITH A SINGLE ACTIVE LATENT

The per-token computational graph of the vertical latent. Consider an autoregressive transformer of depth N and width d, processing a token stream $x_1, x_2, \ldots$. At timestep t, the model maps the input token x_t to a distribution over the next token x_{t+1} through a *single* latent vector $h_t^\ell \in \mathbb{R}^d$, initialized from the token embedding $e_t = h_t^0 = E\,x_t$ and refined across the N layers. Layer $\ell \in \{1, \ldots, N\}$ updates the latent with a pre-norm attention block followed by an MLP block:

$$
\begin{aligned}
q_t^\ell,\, k_t^\ell,\, v_t^\ell &= W_q^\ell \hat{h},\; W_k^\ell \hat{h},\; W_v^\ell \hat{h} \\
\bar{h}_t^\ell &= h_t^{\ell-1} + W_o^\ell\, \mathrm{Attn}\big(q_t^\ell,\, \mathcal{M}_t^\ell\big) \\
h_t^\ell &= \bar{h}_t^\ell + \mathrm{MLP}^\ell\big(\mathrm{Norm}(\bar{h}_t^\ell)\big)
\end{aligned}
$$

where $\hat{h} = \mathrm{Norm}(h_t^{\ell-1})$ and $\mathcal{M}_t^\ell = \{(k_s^\ell, v_s^\ell)\}_{s \le t}$. Throughout, $\mathrm{Norm}(\cdot)$ denotes RMSNorm.

The keys and values the latent produces, (k_t^ℓ, v_t^ℓ), are appended to the layer-ℓ memory or KV cache $\mathcal{M}^\ell$ to be read at future timesteps; this is a persistent memory, written once and never subsequently revised. After the final layer, the latent is decoded as $p(x_{t+1} \mid x_{\le t}) = \mathrm{softmax}\big(W_U\,\mathrm{Norm}(h_t^N)\big)$.

In this graph a single fixed-width latent h_t plays every role at once: it forms the query that reads the memory, it writes the keys and values that are the memory, it is the substrate transformed by the MLP at every layer, and at the top it is the vector decoded into the output distribution. All computation is borne by this single active latent at each timestep, while the memory $\mathcal{M}$ serves only as its persistent, read-only projection for later retrieval.

The recurrent view of the vertical latent. A recent line of work on latent reasoning & continuous language modeling extends the per-token computation across timesteps (Hao et al., 2024; Shen et al., 2025; Zeng et al., 2026). Rather than collapsing the latent to a discrete token at end of each step, the final-layer state h_{t-1}^N of previous timestep is fed directly as input to the first layer of next timestep,

$$
h_t^0 \;=\; h_{t-1}^N, \tag{1}
$$

so that no re-encoding through the vocabulary is required. The motivation is that decoding forces the dense, d-dimensional latent through a discrete token carrying at most $\log_2 V$ bits, for a vocabulary of size V – discarding most of what the latent had computed. Under this view the vertical latent becomes a single running RNN state: one vector, updated at every timestep, that must carry forward all information the model will later need. Under this abstraction, a standard transformer is the special case where this state is reset—decoded and re-encoded—at every step.

3.2 THE LATTICE OF LATENTS

How can we increase the computational capacity of this single active latent? Two options present themselves: widen the latent by increasing its dimension d, or add more latents. We study the second and in our experiments compare it directly against naively widening the single latent. Concretely, we associate each Lattice layer with an additional latent vector that runs *horizontally* across timesteps:

at the first timestep it begins from a learned initial state, and at every timestep thereafter it is an updated version of the same layer's latent from the previous timestep. With one such actively running latent at every Lattice layer, the latents form a lattice – the vertical latent advancing through depth at each timestep, and a horizontal latent advancing through time at each layer—which gives the architecture its name. In what follows, we first describe how each layer's horizontal latent supports the vertical latent in sharing its computational burden, and then give the update mechanism by which the horizontal latents evolve across time.

3.2.1 How the horizontal latents act

Let $r_t^\ell \in \mathbb{R}^{d_r}$ be the horizontal latent at layer ℓ and timestep t, with a width d_r chosen independently of the vertical width d. A standard transformer layer drives its computation from a single input, $h_t^{\ell-1}$ (together with the KV cache written by earlier timesteps). **In the Lattice**, layer ℓ has *two* inputs – the vertical latent $h_t^{\ell-1}$ arriving from below, and the horizontal latent r_t^ℓ, formed by updating the same layer's state from the previous timestep, r_{t-1}^ℓ. We share the $r_{t-1}^\ell \to r_t^\ell$ update procedure in the next section. Both inputs jointly drive the layer's attention and MLP, and together they determine both what the layer adds to the vertical stream—the residual update producing h_t^ℓ—and the memory update, the keys and values it writes into the layer's memory $\mathcal{M}^\ell$ for future timesteps to read. Finally, the topmost Lattice layer's horizontal latent additionally modulates the decoding step that produces the next-token distribution. We specify each of these now – in attention, in the MLP, and in decoding (check Fig. 2). Before use, the horizontal latent is normalized to $\hat{r}_t^\ell = \mathrm{Norm}(r_t^\ell)$.

Conditioning Attention via FiLM. The horizontal latent conditions the query, key, and value the vertical latent ($\hat{h} = \mathrm{Norm}(h_t^{\ell-1})$) forms, rescaling & shifting each by a FiLM transformation (Perez et al., 2017) of readout $\hat{r}_t^\ell$:

$$q_t^\ell = \left(W_q^\ell \hat{h}\right) \odot \left(1 + \gamma_q(\hat{r}_t^\ell)\right) + \beta_q(\hat{r}_t^\ell)$$
$$\gamma_{(\cdot)}(\hat{r}_t^\ell) = \tanh\left(W_{(\cdot)}^\gamma \hat{r}_t^\ell\right) \qquad \beta_{(\cdot)}(\hat{r}_t^\ell) = W_{(\cdot)}^\beta \hat{r}_t^\ell$$

and likewise for k_t^ℓ and v_t^ℓ. The multiplicative gain $1 + \gamma$ is bounded to $(0, 2)$, while β is an additive shift; each has its own zero-initialized weight matrix per stream. Since the modulated k_t^ℓ, v_t^ℓ are exactly what the layer writes to $\mathcal{M}^\ell$, the horizontal latent shapes both memory read and write. These QKV values then drive the attention operation, whose output is added as a residual to the vertical latent, giving $\bar{h}_t^\ell$ – input to the MLP.

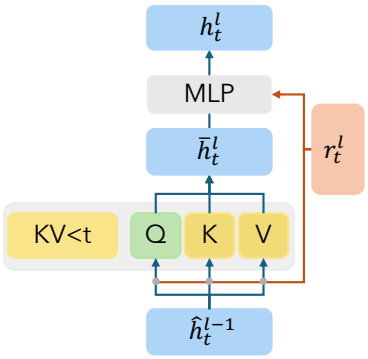


Figure 2: **How the horizontal latent acts.** The explicit recurrent latent $\hat{r}_t^\ell$ conditions the layer's QKV and MLP computations; the topmost horizontal latent also contributes to output decoding (not shown in this figure).

Modulating MLP. Horizontal latent shifts the gate and up pre-activations of the layer's SwiGLU MLP via additive shifts $\beta_{\text{gate}}, \beta_{\text{up}}$, each having its own zero-initialized weight matrix,

$$m = \mathrm{Norm}(\bar{h}_t^\ell) \qquad \text{gate} = W_{\text{gate}}^\ell m + \beta_{\text{gate}}(\hat{r}_t^\ell)$$
$$\text{up} = W_{\text{up}}^\ell m + \beta_{\text{up}}(\hat{r}_t^\ell)$$

The MLP block output $W_{\text{down}}^\ell\big(\mathrm{SiLU}(\text{gate}) \odot \text{up}\big)$ is then added as a residual to $\bar{h}_t^\ell$, giving h_t^ℓ.

Modulating Decoding. The horizontal latent's readout from the last carry-bearing layer, $\hat{r}_t^{\text{top}}$, joins the final vertical latent h_t^N in a shared readout space, and the next token is decoded from their sum:

$$p(x_{t+1} \mid x_{\le t}) = \mathrm{softmax}\Big(W_x\big(P_v \mathrm{Norm}(h_t^N) + P_r \hat{r}_t^{\text{top}}\big)\Big)$$

Here P_v and P_r are learned linear maps projecting the vertical latent and the horizontal latent, respectively, into a shared readout space, before W_x maps that space to vocabulary logits; P_r is zero-initialized, like every other injection in this section. At initialization, the horizontal latent contributes nothing: attention and the MLP reduce to the standard transformer layer, and the output is decoded from the vertical latent alone. It earns its influence only through training.

3.2.2 How the horizontal latents are updated

We divide the transformer's layers into three blocks: an early block (E), a late block (D), and an intermediate *Lattice* block (L) in between. Horizontal latents are associated only with the layers of

L; we ablate the size of L later. We now describe the recurrent state update for r_t^ℓ for $\ell \in L$ as:

$$r_t^\ell = U\big(r_{t-1}^\ell;\ \ h_t^{\ell-1},\ \ e_t,\ \ r_{t-1}^{\text{top}}\big)$$

The base update depends on the horizontal latent's own history, r_{t-1}^ℓ, and the incoming vertical latent, $h_t^{\ell-1}$. Input injection additionally supplies the raw token embedding e_t, while the Jacobi refiner supplies feedback from the top of the Lattice one timestep back, r_{t-1}^{top}. Same-timestep communication from lower horizontal latents is handled separately by vstream, which enriches the readout without entering the temporal recurrence. As noted in the introduction, *our recurrent latent already serves a different role than the recurrent state in Mamba-style models;* furthermore, *these enhancements set the Lattice apart* from hybrid Transformer–SSM Dong et al. (2024); Lieber et al. (2024).

We unpack these mechanisms one at a time below and in Fig. 3.

Affine update from r_{t-1}^ℓ to r_t^ℓ. To ensure parallel teacher-forced training with every position of a sequence trained in parallel, the recurrence updating r_t^ℓ across time must itself be computable for all T timesteps at once, not by a token-by-token loop. To retain an exact, single-pass associative scan, we require the transition to remain affine in r_{t-1}^ℓ. Inspired by linear attention and Mamba-style state-space models, we use an affine recurrence whose coefficients depend only on the current input, never on the state itself. Writing $n_t^\ell = \text{Norm}_{\text{in}}(h_t^{\ell-1})$ for this term's own normalized read of the vertical latent,

$$\begin{aligned}
\Delta_t^\ell &= \text{softplus}\big(W_\Delta^\ell n_t^\ell + b_\Delta^\ell\big)\\
g_t^\ell &= \exp\big(-\exp(a^\ell)\odot \Delta_t^\ell\big)\\
c_t^\ell &= \text{SiLU}\big(W_r^\ell n_t^\ell\big)\\
b_t^\ell &= (1-g_t^\ell)\odot c_t^\ell\\
r_t^\ell &= g_t^\ell \odot r_{t-1}^\ell + b_t^\ell
\end{aligned}$$

Here r_{-1}^ℓ is a learned initial state, and a^ℓ, b_Δ^ℓ are learned per-channel parameters (a log-timescale and its softplus bias, respectively) excluded from weight decay, following Mamba-2 Dao & Gu (2024) practice. Because g_t^ℓ and b_t^ℓ are functions of n_t^ℓ alone and never of r_{t-1}^ℓ, the transition is exactly affine: writing each timestep's map as the pair (g_t^ℓ, b_t^ℓ), any two such maps at positions A and B (with B the later position of the two) compose associatively via

$$(g_A, b_A)\oplus(g_B, b_B) = (g_B\odot g_A,\ \ g_B\odot b_A + b_B)$$

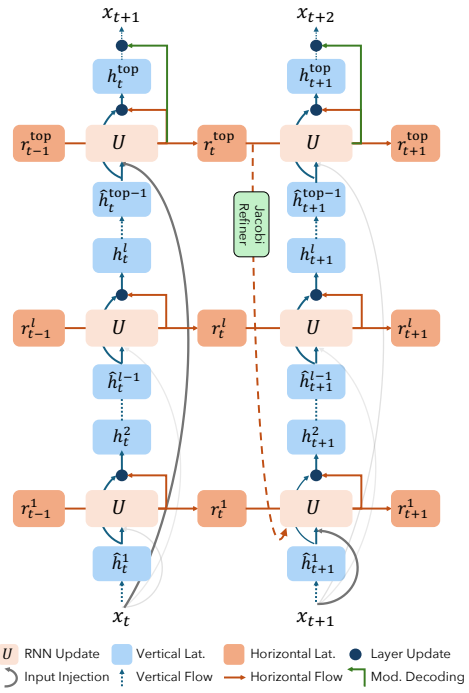


Figure 3: **Horizontal latent update.** On top of the Mamba-style affine update, we further introduce input injection, vstream (not in figure) and cross-timestamp non-linear flow. See webpage for animations.

A single Hillis–Steele scan computes all T states in $O(\log T)$ parallel depth, rather than a $O(T)$ sequential sweep. Hillis–Steele uses $O(T\log T)$ total work; a work-efficient associative scan can achieve the same $O(\log T)$ depth with $O(T)$ work. The readout used downstream is $\hat{r}_t^\ell = \text{Norm}(r_t^\ell)$.

Experiments with this update rule are labeled as **base-update($\mathbf{d}, \mathbf{d_r}$)**, where d and d_r are dimensions of the vertical and horizontal latents respectively. This affine base-update formulation can *integrate* history across time but, on its own, cannot *iterate* nonlinear computation. We therefore study three complementary mechanisms: input-inj and vstream add expressivity within a timestep while preserving the parallel scan, while the Jacobi refiner adds a genuinely nonlinear pathway across time at the cost of the single-pass guarantee.

Adding the input embedding to the horizontal latent. The base update only ever sees $h_t^{\ell-1}$, the vertical latent arriving at layer ℓ. Thus, the information available to the horizontal latent is bottlenecked by what the vertical latent preserves. We show ablations where we make the vertical latent much smaller in Fig. 4. Furthermore, the vertical latent may already have discarded information

about the raw token identity by layer ℓ. Let $e_t := h_t^0$ be the raw layer-0 input embedding with its own normalized read $\hat{e}_t = \mathrm{Norm}_e(e_t)$. This **input-inj** thread injects a (zero-initialized) contribution from $\hat{e}_t$ directly into both pre-activations of the update:

$$\Delta_t^\ell = \mathrm{softplus}\big(W_\Delta^\ell n_t^\ell + W_{e,\Delta}^\ell \hat{e}_t + b_\Delta^\ell\big) \qquad c_t^\ell = \mathrm{SiLU}\big(W_r^\ell n_t^\ell + W_{e,c}^\ell \hat{e}_t\big)$$

with $W_{e,\Delta}^\ell, W_{e,c}^\ell$ each zero-initialized. The recurrence $r_t^\ell = g_t^\ell \odot r_{t-1}^\ell + b_t^\ell$ is otherwise unchanged: since Δ_t^ℓ and c_t^ℓ remain functions of this timestep's inputs alone (n_t^ℓ and now $\hat{e}_t$), never of r_{t-1}^ℓ, the transition is still exactly affine, and the same parallel scan applies unmodified.

Enhancing layer-wise nonlinearity of the horizontal latent. At layer ℓ, the readout $\hat{r}_t^\ell$ — built from the base update and input-inj — carries no information from a lower Lattice layer's horizontal latent, except whatever survived passing back through the vertical latent. This again ties what the horizontal latent can access at a given timestep to whatever the vertical latent's capacity can store.

vstream eases this constraint by threading a second, purely depth-wise state $v_t^\ell \in \mathbb{R}^{d_r}$ bottom-to-top across the Lattice, seeded at $v_t^{\ell_0 - 1} := 0$ at the first Lattice layer ℓ_0:

$$v_t^\ell = v_t^{\ell-1} + W_{\mathrm{up}}^\ell \mathrm{SiLU}\Big(W_{\mathrm{dn}}^\ell \big[\mathrm{Norm}(\hat{r}_t^\ell),\, \mathrm{Norm}(v_t^{\ell-1})\big]\Big),$$

with W_{up}^ℓ zero-initialized, so $v_t^\ell \equiv v_t^{\ell-1}$ at init. This implements same-timestep communication from lower Lattice layers separately from the recurrent update U, which acts along temporal direction. Because v_t^ℓ is a pointwise function of same-timestep quantities alone, it introduces no new dependency across time, only a nonlinear one across depth, and the base recurrence above is untouched. When vstream is enabled, the enriched readout

$$\tilde{r}_t^\ell := \hat{r}_t^\ell + \mathrm{Norm}(v_t^\ell)$$

replaces $\hat{r}_t^\ell$ wherever it steers attention or the MLP in Sec. 3.2.1. It is never used as an input anywhere in the update itself: the temporal recurrence reads and writes r_t^ℓ, never the enriched readout $\tilde{r}_t^\ell$.

Cross-timestep nonlinear update. Both additions so far—input-inj and vstream—increase expressivity using information available at the current timestep: the incoming vertical latent $h_t^{\ell-1}$, the raw embedding e_t, and, through vstream, information propagated from lower Lattice layers. Neither mechanism makes the temporal recurrence's coefficients depend nonlinearly on a past horizontal state, so the parallel scan across time is preserved exactly. What these terms cannot do is route information the other way through depth: they never feed a *higher* layer's latent — the representation that has undergone the deepest, most nonlinear processing in the stack — back down to enrich a *lower* layer's input at the next timestep. This is straightforward at inference or under sequential fine-tuning (a token-by-token rollout), but it breaks the parallelism of teacher-forced pretraining, since the top layer's latent is not yet available when the bottom layer updates. We resolve this with the *Jacobi refiner* (Zeng et al., 2026; Wu et al., 2026), which adds precisely this top→bottom pathway — folding the higher layers' more deeply processed representation into the bottom layer's update and thereby increasing the expressivity of its input – the one deliberate exception to the single-pass scan. In contrast to Zeng et al. (2026); Wu et al. (2026), which use Jacobi iteration to parallelize the continuous-thought recurrence that feeds the final-layer *vertical* latent back as the next timestep's input (Eq. 1), we apply the Jacobi refiner to the *horizontal* latents:

Run the entire Lattice block once, bottom to top, exactly as before; call this sweep $j{=}0$, producing $r_t^{\ell,(0)}$ at every layer and timestep. For each subsequent sweep $j = 1, \ldots, J{-}1$, take the *top* Lattice layer's latent from the *previous* sweep, one timestep back, normalize it, and pass it through a zero-initialized SwiGLU:

$$N_t^{(j)} = W_{\mathrm{down}}\big(\mathrm{SiLU}(W_{\mathrm{gate}}\hat{z}_t^{(j-1)}) \odot (W_{\mathrm{up}}\hat{z}_t^{(j-1)})\big) \qquad \hat{z}_t^{(j-1)} = \mathrm{Norm}\big(r_{t-1}^{\mathrm{top},(j-1)}\big)$$

with the shifted feedback input set to zero at the first timestep and W_{down} zero-initialized. Sweep j then reruns the entire base recurrence unchanged at every layer *except* the bottom Lattice layer ℓ_0, whose bias absorbs this correction:

$$b_t^{\ell_0,(j)} := (1 - g_t^{\ell_0,(j)}) \odot \big(c_t^{\ell_0,(j)} + N_t^{(j)}\big).$$

As in the base update, this bias enters the affine recurrence $r_t^\ell = g_t^\ell \odot r_{t-1}^\ell + b_t^\ell$. This is the mechanism realizing the r_{t-1}^{top} term in U's signature — concretely $r_{t-1}^{\mathrm{top},(j-1)}$, the top layer's latent one timestep

back and one sweep back, injected only at the bottom layer. Because $N_t^{(j)}$ depends only on sweep $j-1$'s fully-resolved output, sweep j's own scan remains exactly affine and parallel across time, just as before. What is lost is the single-pass guarantee itself: composed across J sweeps, the deepest layer's state at one timestep can now shape the shallowest layer's state at a later timestep through a genuinely nonlinear path, at the cost of J serial sweeps over the sequence instead of one, with every sweep remaining token-parallel rather than using a sequential token-by-token unroll. At inference, the top-layer state from the preceding token is already available, so the exact recurrent architecture can be evaluated with one token-by-token pass and no Jacobi refinement sweeps; we separately evaluate this rollout and multi-sweep inference in our experiments.

4 Experiments

4.1 Standard Language Modeling & Downstream Experiments

We pretrain 400M–600M-parameter LLLMs and matched Transformer baselines on the FineWeb-Edu corpus at four token-budget scales – 10B, 20B, 30B tokens – with the larger budgets overtraining the models relative to the Chinchilla-optimal compute allocation for their parameter counts. We evaluate held-out perplexity on both in-domain (FineWeb-Edu validation) and a suite of out-of-domain corpora (Pile, the OpenAI and Standard LAMBADA variants, and Nemotron-CC) to test generalization beyond the pretraining distribution. We additionally report zero- and five-shot downstream accuracy and finetuning transfer in Appendix Sec. A.4.

Standard language modeling Table 1 compares language-modeling quality across 400M–600M models, all pretrained on 20B FineWeb-Edu tokens. At the 400M scale, LLLM $(945, 256)$ matches Base-24L in parameters and compute (406.5 M / 0.72 GFLOP/token vs. 406.7 M / 0.71) yet lowers in-domain perplexity from 15.41 to 14.55 (-5.6%), with consistent 3–11% gains across every out-of-domain set. Adding `input-injection + vstream` – trading vertical width $945 \to 908$ to keep the parameter and FLOP budget fixed – improves further, most on out-of-domain data (including the best Pile in the table, 24.10). Jacobi refinement adds nonlinear top-to-bottom, cross-timestep feedback and yields the strongest 400M result under three-sweep evaluation: LLLM-410 J3 attains the best FineWeb-Edu (14.03), LAMBADA, and Nemotron-CC scores, using 1.85 GFLOP/token for both training and inference. Exact sequential rollout of the same checkpoint removes the Jacobi sweeps and reduces inference compute to 0.72 GFLOP/token, but increases FineWeb-Edu perplexity to 14.73. Prior works Zeng et al. (2026) show that training for random Jacobi iterations instead of fixed (we trained for fixed three iterations) removes sequential rollout discrepancy at no cost.

The trend holds at 600M, where LLLM $(1024, 1024)$ is best within the 600M block on every metric (FineWeb-Edu 13.70), outperforming both width-scaled Base-24L baselines and the depth-scaled Base-40L, the latter at nearly identical compute (1.10 vs. 1.11 GFLOP/token). Distributing a fixed budget across the vertical and horizontal latents thus outperforms spending it on width or depth alone. The efficiency is starkest across scales: the 405.6 M `input-injection + vstream` model already matches the 609.1 M Base-40L on FineWeb-Edu (14.40) and beats it on Pile and both LAMBADA variants—using 33% fewer parameters and 35% less inference compute—trailing only on Nemotron-CC, while the 406.9 M J3 model under three-sweep evaluation surpasses Base-40L on all five

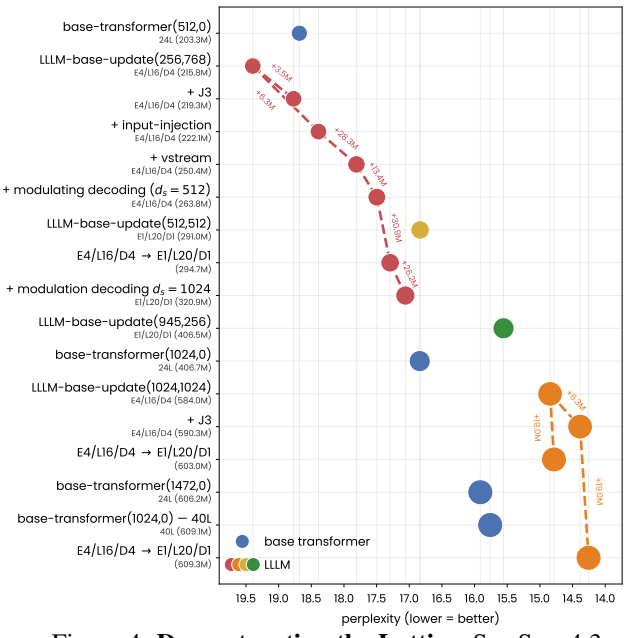


Figure 4: **Deconstructing the Lattice.** See Sec. 4.3

datasets, albeit at higher inference compute. Similar performance holds at 30b as shown in Appendix Tab. 6 and 10b as in Fig. 4.

4.2 The Lattice of Latents Complements Temporal Computation

The Lattice of Latents is complementary to methods that allocate additional computation along the temporal axis. Rather than concentrating this computation in a single recurrent latent, the Lattice distributes it across persistent layer-wise horizontal latents. We therefore combine the Lattice with two alternative temporal-computation strategies—pause tokens (shown in Appendix Sec. A.4) and latent thoughts–while keeping the overall parameter scale approximately fixed.

Lattice with Latent Thoughts. PonderLM-2 propagates the vertical latent from the final layer of one timestep to the first layer of the next (Eq. 1); the Lattice is orthogonal to this and can be layered on top. Table 2 adds the Lattice to PonderLM-2-style latent-thought pretraining. A 22-layer PonderLM-2 with the Lattice (`base-update` only) already nearly matches the 40-layer PonderLM-2 baseline, and adding `input-inj` and `vstream` pushes it past that baseline – even as the horizontal-latent width is shrunk to hold the parameter budget—*while using 18 fewer layers, and hence less KV cache*. Layer-wise latent capacity thus complements the temporal computation of latent thoughts. Since PonderLM-2 already does Jacobi iterations, addition of horizontal latent Jacobi refiner should only add performance benefits at no almost flops / parameter cost – future work.

Approach	Hor. Latent Width (d^r)	PPL ↓
PonderLM-2 (40 layers) (609.0M)	1024	13.604
PonderLM-2 + Lattice *(22 layers total; 20-layer Lattice)*		
base-update (603.0M)	1024	13.617
base-update + input-inj (608.9M)	884	13.525
base-update + input-inj + vstream (609.1M)	752	**13.498**

Table 2: **Matched-scale PonderLM-2 comparison.** Incorporation of Lattice results in matched performance with 18 less layers and much smaller KV-Cache.

4.3 Deconstructing the Lattice

Figure 4 traces perplexity against parameters as we build up the Lattice, all at the 10B-token budget. Starving the vertical latent is costly: shrinking d_v from 1024 to 256 raises perplexity from 16.84 to 19.40—worse even than a smaller d_v=512 no-carry transformer (18.68). Our horizontal-latent mechanisms progressively recover this loss: input injection, vstream, and the decoding head bring the d_v=256 model down to 17.06, nearly matching the full-width $(1024, 0)$ base transformer while using much fewer parameters. Rebalancing capacity between the two latents does better still—a $(512, 512)$ LLLM already edges out the $(1024, 0)$ base (16.83 vs. 16.84) at only 71% of its parameters, and our headline $(945, 256)$ model is the best 400M-scale model overall (15.56), surpassing not only the same-size base but also the $1.5\times$ larger 40-layer transformer (15.76), an effective parameter multiplier of $1.6\times$. *These results show that* how *a fixed budget is divided between the vertical and horizontal latents is itself a performance lever*, and we observe the same trends at the 600M scale.

5 Conclusion

We introduced the *Lattice of Latents*, which augments the standard vertical latent with persistent, layer-wise horizontal latents that participate in attention, memory formation, MLP computation, and decoding. We use an affine horizontal update that preserves token-parallel teacher-forced training, and introduce several enhancements that increase the expressivity of these latents while retaining token-parallel computation. Across language modeling, downstream evaluation, and finetuned mathematical reasoning, LLLMs consistently improve over parameter- and compute-matched Transformer baselines. At the 20B-token scale, a 405.6M-parameter LLLM matches a 609.1M-parameter, 40-layer Transformer on FineWeb-Edu while using 33% fewer parameters and 35% less inference compute. The Lattice also complements pause-token training and latent-thought pretraining. Overall, our results suggest that increasing active computational latent capacity is a promising direction, distinct from using recurrent state primarily as a sequence-mixing or memory mechanism.

REFERENCES

Ziyang Cai, Xingyu Zhu, Yihe Dong, Yinghui He, and Sanjeev Arora. T2MLR: Transformer with temporal middle-layer recurrence. In *Workshop on Latent & Implicit Thinking – Going Beyond CoT Reasoning*, 2026. URL https://openreview.net/forum?id=fQbk1EQWBO.

Federico Danieli, Pau Rodriguez, Miguel Sarabia, Xavier Suau, and Luca Zappella. Pararnn: Unlocking parallel training of nonlinear rnns for large language models, 2025. URL https://arxiv.org/abs/2510.21450.

Tri Dao and Albert Gu. Transformers are ssms: Generalized models and efficient algorithms through structured state space duality, 2024. URL https://arxiv.org/abs/2405.21060.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers, 2019. URL https://arxiv.org/abs/1807.03819.

Xin Dong, Yonggan Fu, Shizhe Diao, Wonmin Byeon, Zijia Chen, Ameya Sunil Mahabaleshwarkar, Shih-Yang Liu, Matthijs Van Keirsbilck, Min-Hung Chen, Yoshi Suhara, Yingyan Lin, Jan Kautz, and Pavlo Molchanov. Hymba: A hybrid-head architecture for small language models, 2024. URL https://arxiv.org/abs/2411.13676.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020.

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, 2025. URL https://arxiv.org/abs/2502.05171.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces, 2024. URL https://arxiv.org/abs/2312.00752.

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.

Zeyi Huang, Xuehai He, LiLiang Ren, Yiping Wang, Baolin Peng, Hao Cheng, Shuohang Wang, Pengcheng He, Jianfeng Gao, Yong Jae Lee, and Yelong Shen. Latent recurrent transformer: Architecture exploration, training strategies, and scaling behavior, 2026. URL https://arxiv.org/abs/2605.26797.

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

Mayank Mishra, Shawn Tan, Ion Stoica, Joseph Gonzalez, and Tri Dao. M^2rnn: Non-linear rnns with matrix-valued states for scalable language modeling, 2026. URL https://arxiv.org/abs/2603.14360.

Giovanni Monea, Nathan Godey, Kianté Brantley, and Yoav Artzi. The state-prediction separation hypothesis, 2026. URL https://arxiv.org/abs/2607.01218.

Michael C. Mozer, Shoaib Ahmed Siddiqui, Danny Sawyer, Sunny Sanyal, and Rosanne Liu. Recirculation, 2026. URL https://arxiv.org/abs/2608.17981.

James O' Neill and Fergal Reid. Looped latent attention: Cross-loop kv compression for looped transformers, 2026. URL https://arxiv.org/abs/2607.15456.

Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Ngoc-Quan Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. The lambada dataset: Word prediction requiring a broad discourse context. In *Proceedings of the 54th annual meeting of the association for computational linguistics (volume 1: Long papers)*, pp. 1525–1534, 2016.

Ethan Perez, Florian Strub, Harm de Vries, Vincent Dumoulin, and Aaron Courville. Film: Visual reasoning with a general conditioning layer, 2017. URL https://arxiv.org/abs/1709.07871.

Zhenyi Shen, Hanqi Yan, Linhai Zhang, Zhanghao Hu, Yali Du, and Yulan He. Codi: Compressing chain-of-thought into continuous space via self-distillation, 2025. URL https://arxiv.org/abs/2502.21074.

Julien Siems, Timur Carstensen, Arber Zela, Frank Hutter, Massimiliano Pontil, and Riccardo Grazzi. Deltaproduct: Improving state-tracking in linear rnns via householder products, 2025. URL https://arxiv.org/abs/2502.10297.

Dan Su, Kezhi Kong, Ying Lin, Joseph Jennings, Brandon Norick, Markus Kliegl, Mostofa Patwary, Mohammad Shoeybi, and Bryan Catanzaro. Nemotron-cc: Transforming common crawl into a refined long-horizon pretraining dataset. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2459–2475, 2025.

Victor Conchello Vendrell, Arnau Padres Masdemont, Niccolò Grillo, Jordi Ros-Giralt, Arash Behboodi, and Fabio Valerio Massoli. Memory-efficient looped transformer: Decoupling compute from memory in looped language models, 2026. URL https://arxiv.org/abs/2605.07721.

Xi Wang, Ziyang Cai, Zheng Zhan, Harry Dong, Ying Fan, Gustavo de Rosa, Tim Pearce, and John Langford. Full-bandwidth transformer, 2026. URL https://arxiv.org/abs/2608.08888.

Haoyi Wu, Zhihao Teng, and Kewei Tu. Parallel continuous chain-of-thought with jacobi iteration, 2026. URL https://arxiv.org/abs/2506.18582.

Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated delta networks: Improving mamba2 with delta rule, 2025. URL https://arxiv.org/abs/2412.06464.

Boyi Zeng, He Li, Shixiang Song, Yixuan Wang, Zitong Wang, Ziwei He, Xinbing Wang, and Zhouhan Lin. Ponderlm-2: Pretraining llm with latent thoughts in continuous space, 2026. URL https://arxiv.org/abs/2509.23184.

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, 2026. URL https://arxiv.org/abs/2510.25741.

A Appendix

A.1 AI Use Statement

This project used Claude Code as a coding and analysis assistant throughout development. Concretely, it (i) wrote and debugged the implementation of the Lattice of Latents and of the baseline models; (ii) prepared, submitted, and tracked our pretraining and evaluation jobs on the compute cluster; (iii) helped interpret and sanity-check experimental results and (iv) refine the author written manuscript for clarity. The research questions, the architectural hypotheses, and the overall experimental design were formulated by the authors rather than the assistant. No datasets were synthesized by AI; datasets are borrowed from prior works which we use unmodified. Bibliographic entries were compiled by hand. Generated code was read and tested; and every written content the assistant touched was proof-read and modified.

A.2 Reproducibility

We provide training setup and hyper-parameter details, and code will be released upon acceptance.

A.3 Extended Related Work

Depth recurrence and latent reasoning. Prior work increases a language model's computation either along depth or across time. Depth-based approaches add layers or repeatedly apply a shared stack (Dehghani et al., 2019; Geiping et al., 2025; Zhu et al., 2026). In standard attention stacks, additional layers introduce additional per-layer KV caches; looped models may likewise retain iteration-specific caches, although recent work instead shares or compresses state across loops (Vendrell et al., 2026; Neill & Reid, 2026). Thus, additional depth computation is often coupled to additional persistent memory.

Continuous-latent methods instead carry a single evolving hidden state through additional latent steps before producing the next text token. Coconut feeds the previous hidden state back as the next input, CODI distills continuous hidden-state reasoning from an explicit chain-of-thought teacher, and PonderLM-2 pretrains this behavior (Hao et al., 2024; Shen et al., 2025; Zeng et al., 2026). When latent thoughts are represented as additional causal positions, they also create additional KV entries. Moreover, the dependency between consecutive thoughts prevents ordinary single-pass teacher forcing; PonderLM-2 and PCCoT recover token parallelism using Jacobi-style iterative refinement (Zeng et al., 2026; Wu et al., 2026). LLLM differs from both depth- and time-recurrence approaches by distributing active computation across multiple persistent, layer-indexed latents, rather than repeatedly updating only one computational stream. These horizontal states add fixed-size recurrent state without adding tokenwise entries to the attention KV cache.

Parallel recurrent sequence models. Our affine horizontal update builds on scan-parallel recurrent models. Mamba uses an input-dependent gated recurrence that can be evaluated with a parallel scan (Gu & Dao, 2024; Dao & Gu, 2024). Gated DeltaNet introduces delta-rule state updates, while DeltaProduct composes multiple Householder-style corrections into a more expressive linear transition (Yang et al., 2025; Siems et al., 2025). Their common property is that the transition remains linear or affine in the previous state, permitting parallel evaluation across time.

LLLM uses the same algebraic property for a different architectural purpose. In hybrid Mamba–Transformer models, recurrent state remains internal to an SSM sequence-mixing branch, which may replace attention at selected layers, as in Jamba, or operate alongside attention heads, as in Hymba (Lieber et al., 2024; Dong et al., 2024). In LLLM, the horizontal state instead remains an explicit companion computational latent: it conditions softmax attention and the resulting KV-cache writes, modulates the MLP, and contributes to decoding. Work on nonlinear recurrent training is complementary. ParaRNN applies Newton iterations to parallelize general nonlinear recurrences, while M^2 RNNs combine nonlinear updates with matrix-valued states (Danieli et al., 2025; Mishra et al., 2026). We retain an affine temporal update and increase expressivity through its inputs and readouts, with optional iterative refinement for nonlinear feedback across time.

Approach	Params (M)	LAMBADA		ARC		WG	PIQA	HS	SciQ	RACE	Avg. ↑	Δ ↑
		OpenAI	Standard	Easy	Challenge							
0-shot accuracy												
400M-parameter models												
Base-24L (width $d_v = 1024$)	407	33.0	24.9	53.7	27.8	50.6	66.8	40.2	81.4	30.7	45.46	—
LLLM $(945, 256)$	407	35.9	26.9	56.4	29.0	54.2	68.4	42.4	84.4	31.3	47.67	2.21
LLLM $(908, 256)$ re+vs	406	35.1	28.2	56.0	29.6	51.6	68.0	43.6	82.6	31.7	47.38	1.92
LLLM-410 J3 $(945, 256)$	407	38.2	29.6	55.6	30.1	55.0	69.0	44.9	83.7	32.2	48.69	3.23
500M-parameter models												
LLLM $(512, 512)$, 40L	495	35.9	28.9	56.6	27.8	54.5	68.9	44.2	83.4	32.7	48.10	2.64
600M-parameter models												
Base-24L (width $d_v = 1620$)	603	33.1	25.5	54.9	28.8	50.9	66.7	41.5	82.9	30.9	46.14	0.68
Base-24L (width $d_v = 1472$)	606	35.1	27.1	55.6	28.8	52.1	68.0	42.1	84.3	33.0	47.36	1.90
Base-40L (width $d_v = 1024$)	609	35.2	27.4	56.1	29.1	53.3	67.8	42.6	84.1	30.5	47.36	1.90
LLLM $(1024, 1024)$	603	36.0	28.6	57.6	29.5	52.4	69.1	45.7	84.5	33.3	48.53	3.07
5-shot accuracy												
400M-parameter models												
Base-24L (width $d_v = 1024$)	407	25.5	24.1	59.3	30.9	51.5	66.8	39.9	86.5	31.4	46.19	—
LLLM $(945, 256)$	407	27.3	24.4	63.6	30.6	52.2	67.7	42.6	88.4	33.3	47.79	1.60
LLLM $(908, 256)$ re+vs	406	29.8	26.9	65.0	30.8	51.2	67.5	43.1	87.7	31.4	48.16	1.97
LLLM-410 J3 $(945, 256)$	407	29.2	26.3	65.2	33.5	54.3	68.9	45.3	90.2	31.5	49.38	3.19
500M-parameter models												
LLLM $(512, 512)$, 40L	495	29.1	26.8	64.1	31.6	52.3	67.9	43.8	89.2	31.7	48.50	2.31
600M-parameter models												
Base-24L (width $d_v = 1620$)	603	24.4	23.4	62.0	32.4	51.3	67.2	41.6	87.3	30.0	46.63	0.44
Base-24L (width $d_v = 1472$)	606	26.4	23.2	62.8	31.7	52.0	68.3	42.2	89.2	30.4	47.34	1.15
Base-40L (width $d_v = 1024$)	609	26.3	24.1	63.2	31.6	51.1	67.4	43.2	88.6	30.9	47.37	1.18
LLLM $(1024, 1024)$	603	29.3	28.2	65.6	32.9	52.8	68.0	45.3	89.4	32.7	49.37	3.18

Table 3: **Zero- and five-shot downstream task accuracy at the 20B-token FineWeb-Edu pretraining scale.** Models are grouped by parameter scale and shown in the same order under both evaluation settings. All task values are accuracies (%; ↑). LAMBADA results are reported on the OpenAI and Standard variants, while ARC results are reported on ARC-Easy and ARC-Challenge. WG and HS denote WinoGrande and HellaSwag, respectively. Avg. is the reported average across the nine tasks. Within each evaluation setting, Δ denotes reported difference from Base-24L avg.

Concurrent recurrent and multi-stream Transformers. Several concurrent works introduce recurrent or multi-stream pathways into Transformer architectures. Full-Bandwidth Transformer (Wang et al., 2026) feeds the previous token's top-layer state back alongside the next token embedding, while Latent Recurrent Transformer (Huang et al., 2026) reuses a selected high-level representation as recurrent memory for the next token. T^{2}MLR (Cai et al., 2026) similarly routes a deeper representation from the previous token into an earlier layer of the current token through a gated pathway. T^{2}MLR is the closest precedent to our Jacobi refiner, but the recurrent object is a cached vertical representation. Our refiner instead communicates between the top and bottom of a pre-existing lattice of horizontal states, each of which also follows its own layer-local temporal recurrence. The State-Prediction Separation Transformer (Monea et al., 2026) uses separate computation streams for preparing persistent state and predicting the next token. This shares our motivation that a single representation may be burdened by multiple roles, but separates those roles within token computation rather than maintaining a persistent recurrent latent at each layer. Recirculation (Mozer et al., 2026) also sends deep-layer activations back to shallower layers across input steps, but is primarily a training-free, inference-time intervention whose prefill is processed sequentially. In contrast, LLLM is trained end to end, maintains a distinct horizontal latent at each Lattice layer, and preserves token-parallel teacher-forced training for its base update. These approaches are therefore complementary: concurrent work largely adds a single feedback pathway or separates two computational roles, whereas LLLM studies how multiple persistent, layer-wise latents can jointly support the Transformer computation.

A.4 Experimental Results

Scaling to 30B tokens. Building on the 20B-token results in Table 1, Table 6 shows that the benefits of the Lattice persist under substantially longer, post-Chinchilla pretraining on 30b to-

kens. At the matched 400M scale, LLLM $(908, 256)$ with input injection and `vstream` outperforms Base-24L on every reported metric, reducing FineWeb-Edu, Pile, LAMBADA-OpenAI, and LAMBADA-Standard perplexity by 5.4%, 8.9%, 8.1%, and 11.2%, respectively, while improving the two LAMBADA accuracies by 2.90 and 3.40 points. The gains also hold at 600M parameters: LLLM $(1024, 1024)$ improves over the similarly sized Base-40L by 4.4% on FineWeb-Edu, 4.6% on LAMBADA-OpenAI, and 7.3% on LAMBADA-Standard, while increasing their accuracies by 1.79 and 0.75 points; Base-40L remains modestly better on Pile. Thus, the advantages observed at 20B tokens do not disappear with extended pretraining. Moreover, the strong results obtained with different vertical–horizontal allocations suggest that this capacity split is an important tuning axis, and that a more systematic search may yield further improvements.

Downstream task performance. Table 3 reports zero- and five-shot accuracy across nine downstream benchmarks after 20B-token pretraining, complementing the language-modeling evaluation above. At the matched 400M scale, LLLM $(945, 256)$ improves average accuracy over Base-24L from 45.46 to 47.67 in the zero-shot setting ($+2.21$ points) and from 46.19 to 47.79 in the five-shot setting ($+1.60$ points). Input injection and `vstream` further strengthen the five-shot result to 48.16, while Jacobi refinement produces the best 400M performance: 48.69 zero-shot and 49.38 five-shot, improvements of 3.23 and 3.19 points over the matched Transformer, respectively, with gains on every individual task. The same trend holds at 600M parameters, where LLLM $(1024, 1024)$ surpasses the 609M Base-40L Transformer by 1.17 points zero-shot and 2.00 points five-shot. Notably, the 407M J3 model also exceeds Base-40L by 1.33 and 2.01 points, respectively, despite using roughly one-third fewer parameters, although it requires additional Jacobi sweeps.

Finetuning on GSM8K-Aug. Table 4 evaluates how well the 20B-token-pretrained models transfer after finetuning on GSM8K-Aug. At the 400M scale, every LLLM variant outperforms the matched Base-24L Transformer across all five reasoning benchmarks. LLLM-410 J3 achieves the highest mean accuracy, improving from 36.02 to 48.10 ($+12.08$ points), and obtains the strongest GSM8K and SVAMP-1K results. The base-update model performs best on MultiArith and GSM-Hard, while input injection and `vstream` produce the best ASDiv accuracy, suggesting that the different horizontal-update mechanisms provide complementary transfer benefits. The trend persists at the 600M scale: LLLM $(1024, 1024)$ outperforms the 609M Base-40L Transformer on every task and raises mean accuracy from 40.24 to 47.74 ($+7.50$ points), despite using slightly fewer parameters. These results show gains from horizontal latent capacity transfer beyond pretraining perplexity to math reasoning.

Approach	GSM 8K	SVAMP 1K	Multi Arith	ASDiv	GSM Hard
400M-parameter models					
Base-24L ($d_v = 1024$)	29.7	28.4	74.0	41.6	6.4
LLLM $(945, 256)$	44.5	42.0	**90.8**	51.4	**10.8**
LLLM $(908, 256)$ re+vs	43.6	42.6	87.7	**52.0**	10.0
LLLM J3 $(945, 256)$	**44.9**	**45.2**	89.2	51.5	9.7
600M-parameter models					
Base-24L ($d_v = 1620$)	30.0	31.7	79.7	42.9	6.9
Base-40L ($d_v = 1024$)	33.5	32.7	83.3	44.7	7.0
LLLM $(1024, 1024)$	**45.7**	**42.6**	**89.0**	**51.0**	**10.4**

Table 4: **Finetuned mathematical-reasoning Acc.**

Lattice with Pause Tokens. Table 5 shows that the Lattice remains effective alongside pause-token ("thinking") training, and that the two sources of extra computation are complementary. Without pause tokens, the 22-layer LLLM (20-layer Lattice) already improves on the 40-layer Transformer despite using 45% fewer layers. Pause tokens help every model, while Jacobi refinement compounds these gains: the best overall perplexity is obtained with two pause tokens and J3, at the cost of additional refinement passes. Even in the

Approach	Hor. latent width (d^r)	# Pause tokens	PPL ↓	
			J0	J3
40-layer Base Transformer (609.1 M)				
Base Transformer	—	0	15.76	—
Base Transformer	—	$\in \{1, 2\}$	14.08	—
LLLM $(1024, d^r)$ (603 M) *(22 layers total; 20-layer Lattice)*				
base-update	1024	0	14.78	14.25
base-update	1024	1	14.34	13.83
base-update	1024	$\in \{1, 2\}$	14.38	13.73
base-update	1024	2	14.26	13.66
base-update + input-inj	860	1	14.21	—
base-update + input-inj + vstream	734	1	14.18	—

Table 5: **Pause-token and horizontal-update ablations.**

Approach	Params (M)	FineWeb-Edu PPL ↓	Pile PPL ↓	LAMBADA OpenAI		LAMBADA Standard	
				PPL ↓	Acc. ↑	PPL ↓	Acc. ↑
400M-parameter models							
Base-24L (width $d_v = 1024$)	407	14.719	26.518	40.759	34.83	69.346	26.49
LLLM $(945, 256)$	407	14.050	25.081	37.605	37.36	64.314	28.80
LLLM $(908, 256)$ re+vs	406	**13.921**	**24.148**	**37.451**	**37.73**	**61.598**	**29.89**
500M-parameter models							
LLLM $(512, 512)$, 40L	495	13.831	24.203	37.525	39.38	63.721	30.84
600M-parameter models							
Base-24L (width $d_v = 1620$)	603	13.944	26.793	37.404	37.53	62.158	28.31
Base-24L (width $d_v = 1472$)	606	14.033	25.016	38.501	36.79	64.490	29.94
Base-40L (width $d_v = 1024$)	609	13.802	**23.478**	38.123	38.54	63.146	30.16
LLLM $(1024, 1024)$	603	**13.189**	23.818	**36.365**	**40.33**	**58.516**	**30.91**

Table 6: **Language-modeling quality after 30B-token FineWeb-Edu pretraining.** We report perplexity (↓) on FineWeb-Edu, Pile, and the OpenAI and Standard LAMBADA variants, together with LAMBADA last-word accuracy (↑). Models are grouped by parameter scale.

single-pass (J0) setting, `input-inj`
and `vstream` improve the one-pause result while the horizontal-latent width is shrunk to hold the parameter budget, indicating that *the vertical-to-horizontal latent split is an additional lever for improving performance*.

A.5 Training Details

Unless otherwise noted, all models use the same training recipe; only model width and depth, token budget, and Lattice-specific settings vary across runs.

Backbone. We build on a LLaMA-style decoder with the GPT-NeoX tokenizer (vocabulary size $V = 50{,}304$). Layers use pre-RMSNorm ($\epsilon = 10^{-5}$), rotary position embeddings (RoPE) with a context length of 2048, SwiGLU MLPs with intermediate size 2752, and an *untied* output head. Attention uses 16 query heads and 16 key/value heads, each of dimension 64, so the attention width ($16 \times 64 = 1024$) is fixed and decoupled from the residual-stream (vertical-latent) width d_v. We vary $d_v \in \{256, 512, 908, 945, 1024, \dots\}$ and the depth $N \in \{22, 24, 40\}$ to sweep model size. Input embeddings are scaled by $\sqrt{d_v}$.

Optimizer. We use AdamW with β_1=0.9, β_2=0.95, weight decay 0.1, and gradient-norm clipping at 1.0. Following Mamba-2 practice, the recurrence parameters $\mathrm{A_log}$ and $\mathrm{dt_bias}$, the learned initial state r_{-1}, and all biases and normalization weights are excluded from weight decay. All Lattice injection maps—the FiLM γ/β projections, the input-injection and `vstream` maps, the Jacobi refiner's output projection, and the output-head carry projection—are zero-initialized. Consequently, the Lattice model is functionally identical to the corresponding plain Transformer at initialization, and the horizontal latent acquires influence only through training. All other weights use the standard $\mathcal{N}(0, 0.02^2)$ initialization.

Learning-rate schedule. The peak learning rate is 1×10^{-3} and is decayed with a cosine schedule to a floor of 10% of the peak (`cosine_with_min_lr`, `min_lr_rate`=0.1), following a linear warmup over the first 2% of training steps.

Batching and schedule length. We use a global batch size of 512 sequences of length 2048 ($\approx$ 1.05M tokens per step), realized with a per-device batch size of 8 and gradient accumulation of 8 across 8 GPUs. Token budgets correspond to 9,537 steps for 10B tokens, 19,074 for 20B, 28,611 for 30B.

Precision and kernels. Training uses `bfloat16` mixed precision, FlashAttention-2, and gradient checkpointing. For numerical stability, the diagonal gated-linear carry scan and its FiLM application are computed in `fp32`.

Data. We pretrain on FineWeb-Edu, pre-tokenized with the NeoX tokenizer and packed into contiguous 2048-token sequences (dataset shards `fineweb_edu_{10,20,30,50}bt_neox2048`). Perplexity is reported on a deduplicated, contamination-checked, held-out FineWeb-Edu split, with zero-shot evaluation on Pile, LAMBADA, and Nemotron-CC.

Lattice configuration. An LLLM is specified by its early/Lattice/late split $E/L/D$ (e.g., $E1/L20/D1$ or $E4/L16/D4$, with horizontal latents only in the L block), the horizontal-latent width $d_r = \mathrm{round}(\texttt{carry_expand} \cdot d_v)$, groupwise FiLM with `film_gamma_groups`=4, the number of Jacobi sweeps $J \in \{0, 3\}$, and the pause-token budget K. Unless stated otherwise, all headline results use K=0, corresponding to no pause tokens.

Hyperparameter	**Value**
Optimizer	AdamW (β_1=0.9, β_2=0.95)
Weight decay	0.1 (none on norms/biases/A_log/dt_bias/r_{-1})
Peak LR / floor	1×10^{-3} / 10% of peak
LR schedule	cosine-with-min-lr, 2% linear warmup
Grad clip	1.0
Global batch / seq. length	512 / 2048 ($\approx$ 1.05M tokens/step)
Per-device batch $\times$ grad-accum.	8×8 on 8 GPUs
Steps (10/20/30)	9,537 / 19,074 / 28,611
Precision	bf16 (carry scan/FiLM in fp32), FlashAttention-2
Vocab. / context	50,304 / 2048
Heads (Q/KV) $\times$ head dim.	$16/16 \times 64$ (attention width 1024)
MLP intermediate	2752 (SwiGLU)
Embedding scale	$\sqrt{d_v}$

Table 7: Training hyperparameters shared across all runs unless otherwise noted.