VecTur: Vector Turing Machines

Ethan Hall
ethan.hall.phd@gmail.com

Abstract

We introduce VECTUR (Vector Turing Machines), a differentiable, Turing-machine-inspired transition system that represents tape symbols, head position, and finite control as continuous vectors (Turing, 1936). Conceptually, VECTUR continues a well-trodden line of differentiable memory machines (e.g., Neural Turing Machines and Differentiable Neural Computers) (Graves et al., 2014, 2016); our focus is a modern, sparse implementation that avoids dense content-based access at every step. VECTUR maintains a continuous "head on a circle" (S^1) and performs *strictly local*, $2k$-sparse gather/scatter updates via interpolation, encouraging pointer-machine-style computation and mitigating the "memory blurring" failure mode of early dense-access NTMs. VECTUR also supports *deep computation* (Dehghani et al., 2019) by explicitly iterating a learned transition map and using an ACT-style learned halting gate (Graves, 2016) (parameterized by κ); we treat this halting parameterization as a practical heuristic rather than a core novelty claim. We evaluate VECTUR as a drop-in *computational block* inside a Llama-style decoder-only language model (Touvron et al., 2023; Dubey et al., 2024), contrasting it with attention (Vaswani et al., 2017), LSTM-style recurrence (Hochreiter and Schmidhuber, 1997), and differentiable external-memory baselines (Graves et al., 2014, 2016). On small-to-medium open benchmarks for reasoning and language (GSM8K (Cobbe et al., 2021), ARC (Clark et al., 2018), HellaSwag (Zellers et al., 2019), WikiText-103 (Merity et al., 2016)), we find VECTUR improves algorithmic generalization (Kaiser and Sutskever, 2016; Press et al., 2022) at fixed parameter budgets. We additionally introduce COMPGEN—a synthetic program dataset stratified by $(T(n), S(n))$ complexity classes—as a utility benchmark for probing compute allocation. Finally, we propose VECSTUR, a stochastic extension that consumes random tape symbols and targets *randomized algorithms* as a computational resource: VECSTUR outperforms VECTUR on randomized verification tasks (e.g., Freivalds-style matrix product verification (Freivalds, 1977)).

1 Introduction

Modern language models excel at pattern completion yet often struggle to reliably *execute* long algorithmic computations, extrapolate beyond training lengths (Press et al., 2022), or allocate variable compute per input (Graves, 2016; Dehghani et al., 2019). Several lines of work attempt to address these limitations by embedding algorithmic structure into neural systems, including external-memory architectures (Graves et al., 2014, 2016) and adaptive computation mechanisms (Graves, 2016).

We propose VECTUR, a vectorized analogue of a classical Turing machine (Turing, 1936) whose tape, head index, and finite control are represented as continuous vectors and updated by a learned transition map. We do *not* claim to have invented differentiable Turing machines; rather, we revisit this classic idea in a form that better matches modern LLM systems constraints. Classic NTMs (Graves et al., 2014) relied on dense, content-based attention over the entire memory at each step, which is $O(N)$ in memory size and can introduce diffuse "blurring" updates. In contrast, VECTUR maintains a continuous head position on a circular tape (S^1) and enforces *strictly local* access: each

Submitted to 38th Conference on Neural Information Processing Systems (NeurIPS 2024). Do not distribute.

step reads and writes via interpolation over only $2k$ tape indices (sparse gather/scatter), yielding per-step cost independent of tape length and an inductive bias closer to pointer machines (Vinyals et al., 2015). For adaptive depth, VECTUR includes an ACT-style halting mechanism (Graves, 2016); our particular κ parameterization is presented as a practical, input-conditioned control knob rather than a conceptual departure from ACT.

Why this helps for long context. Self-attention provides direct interaction between all token pairs but incurs $O(N^2)$ compute in sequence length N. Linear-time alternatives avoid this quadratic cost, but typically do so by enforcing a *linear progression* of computation through the sequence: in an LSTM, information from the past must be compressed into a fixed-size hidden state, and each new token updates that state once in time order. More recent linear-time memory perspectives (e.g., the MIRAS viewpoint connecting retention, memorization, and online optimization (Behrouz et al., 2025)) likewise maintain and update memory in lockstep with the token stream. In contrast, VECTUR decouples interaction from all-pairs attention by introducing an explicit *Turing index* (the head index I_t) whose motion over the tape is learned. Across transition steps, head motion allows the model to create relationships between *arbitrary* tokens by moving to (or copying from) the corresponding tape locations and combining them through the finite-control state, without computing a dense $N \times N$ similarity matrix. Crucially, this enables *non-linear* token processing: the controller can revisit selected tokens or intermediate scratchpad cells multiple times, in an order determined by the evolving machine state, while leaving the rest of the tape unchanged by construction. This selective persistence—deciding what to write to the tape, what to overwrite, and what to ignore—matches the operational structure of Turing-style computation (state + tape) more directly than recurrence with a single compressed memory vector.

We evaluate VECTUR in a realistic regime by inserting it as a *block* inside a Llama-style decoder-only macro architecture (Touvron et al., 2023; Dubey et al., 2024), replacing the standard attention+MLP block. We compare against alternative blocks: (i) standard attention (Vaswani et al., 2017), (ii) LSTM-style recurrence (Hochreiter and Schmidhuber, 1997), and (iii) differentiable external-memory controllers (Graves et al., 2014, 2016). We focus on small-to-medium models (roughly 10^8 to 10^9 parameters) where architectural inductive bias can materially affect sample efficiency and extrapolation (Kaiser and Sutskever, 2016; Press et al., 2022).

We further introduce COMPGEN, a dataset of generated Python programs grouped into discrete complexity buckets $(T(n), S(n))$ such as $O(n)$/$O(1)$, $O(n \log n)$/$O(1)$, $O(n^2)$/$O(1)$, and $O(n^2)$/$O(n)$. The goal is to probe whether a model can learn to *compute* across increasing n by allocating more steps as needed, rather than memorizing only small n. Finally, we define VECSTUR, which augments the input with stochastic symbols z to emulate randomized computation, and we propose a randomized evaluation suite where randomness yields asymptotic speedups (Freivalds, 1977; Miller, 1976; Rabin, 1980). In particular, we form a data set of matrices $A, B, C \in \mathbb{R}^{n \times n}$ and a target matrix $D \in \mathbb{R}^{n \times n}$ such that $D = AB$ and $D = AC$ with probability $1/2$. We evaluate VECTUR and VECSTUR on this task, and show VECSTUR can exploit stochastic symbols to achieve asymptotic speedups.

Contributions.

- We define VECTUR, a Turing-style transition system with *strictly local*, $2k$-sparse gather/scatter tape access (continuous head on S^1), addressing efficiency and "memory blurring" issues associated with dense-access NTMs (Graves et al., 2014).
- We present a plug-and-play integration of VECTUR as a *computational sub-layer* inside Llama-style decoder-only models, treating iterative algorithmic computation as a composable block rather than a separate retrieval module.
- We define VECSTUR and a randomized computation evaluation suite, highlighting randomness as a computational resource (e.g., Freivalds-style verification (Freivalds, 1977)) rather than mere noise.
- We introduce COMPGEN, a synthetic program dataset labeled by time/space complexity class $(T(n), S(n))$, as a utility benchmark for extrapolation and compute-allocation probing.

Feature	NTM (Graves)	VecTur	VecTur advantage
Addressing	Dense content-based attention over all slots	Local/location-based head movement on tape	**Yes**
Per-step complexity (vs. tape length N_T)	$O(N_T)$ similarity + weighted sum	$O(k)$ gather/scatter (independent of N_T)	**Yes**
Forward activation memory (unrolled T steps)	Stores dense weights $\sim O(TN_T)$	Stores sparse indices/weights $\sim O(Tk)$	**Yes**
State preservation away from head	Many slots updated slightly (drift/blurring risk)	Un-accessed cells are exactly unchanged	**Yes**
Content lookup in 1 step	Native (query by key)	Requires scanning via head movement (worst-case $O(N_T)$ steps)	No
Inductive bias	Random-access / associative recall	Sequential pointer machine / local algorithms (Vinyals et al., 2015)	Depends
Compute allocation / halting	Typically fixed unroll or implicit stopping	Explicit learned halting via κ and gate g_t	**Yes**
Fit as a Transformer block	Redundant global attention inside block	Complements attention with iterative scratchpad dynamics	**Yes**

Figure 1: **NTM vs. VecTur.** NTMs (Graves et al., 2014) provide dense, content-addressable memory access, while VecTur enforces sparse, local tape access with adaptive depth. The rightmost column highlights regimes where VecTur is especially advantageous as a computational block inside attention-based macro-architectures.

2 Related Work

Sequence models and attention. Transformers (Vaswani et al., 2017) and their decoder-only variants power modern LLMs (e.g., GPT-3 (Brown et al., 2020) and Llama-family models (Touvron et al., 2023; Dubey et al., 2024)). Recurrent networks such as LSTMs (Hochreiter and Schmidhuber, 1997) provide a different inductive bias for iterative computation but historically underperform attention-based models at scale on language modeling.

Differentiable memory and neural machines. Neural Turing Machines (NTMs) (Graves et al., 2014) and Differentiable Neural Computers (DNCs) (Graves et al., 2016) integrate external memory with differentiable read/write heads. Our work shares the goal of improving algorithmic behavior, but emphasizes sparse, strictly local access (pointer-machine-style) and composable integration as a modern Transformer block; we include learned halting primarily as an ACT-style compute control mechanism.

2.1 Remark: Neural Turing Machines vs. VecTur

Both NTMs (Graves et al., 2014) and VecTur augment neural computation with an external memory, but they make different design trade-offs for *addressing* (how memory is accessed) and *sparsity* (how much memory is touched per step). NTMs provide content-addressable reads/writes via dense attention over all memory slots; VecTur instead maintains a continuous head position on a tape and performs sparse gather/scatter updates to a small number of adjacent cells (via interpolation), which keeps per-step cost independent of tape length. Dense access is expressive but $O(N)$ in memory size and can induce diffuse "memory blurring" updates when many slots receive small writes; VecTur preserves untouched cells exactly by construction. In Llama-style Transformer blocks, global content-based access is already available through self-attention; VecTur is intended to add an *orthogonal* capability: cheap, iterative state manipulation with persistent scratchpad dynamics.

Adaptive computation. Adaptive Computation Time (ACT) (Graves, 2016) learns when to stop iterating, with later refinements such as PonderNet (Banino et al., 2021). Our learned gate g_t and κ-parameterization should be viewed as an ACT-style variant that provides a simple, input-conditioned control knob for effective depth; we do not position halting as the primary conceptual novelty.

Test-time training and online optimization. Recent work reframes sequence modeling as a form of *online learning* or nested optimization carried out during inference, including end-to-end test-time training for long-context language modeling (Tandon et al., 2025) and the MIRAS framework connecting attention, retention, and online optimization (Behrouz et al., 2025). This line of work motivates the viewpoint that "System 2" computation can be injected *inside* a model by adding inner-loop dynamics as a composable block within the forward pass, rather than only via external deliberation or separate modules.

Randomized algorithms. Randomness can reduce expected runtime for verification and decision problems; canonical examples include Freivalds' randomized matrix product verification (Freivalds, 1977) and probabilistic primality testing (Miller, 1976; Rabin, 1980). VECSTUR is intended as a neural analogue that can exploit stochastic symbols during computation.

3 VecTur: Vector Turing Machines

3.1 Vectorized machine state

Given an input sequence $x \in \mathbb{R}^{N \times d_x}$ (e.g., token embeddings), we define a VECTUR block below. Note that for VECSTUR, we additionally sample a sequence of stochastic symbols $z \in \mathbb{R}^{N_z \times d_x}$ and set the tape length

$$N_T = N + N_z, \tag{1}$$

so that each input symbol and each stochastic symbol can be addressed at least once. In our experiments we use $N_z \approx N$ (so $N_T \approx 2N$).

We define the machine state at step t as a triple (T_t, Q_t, I_t), where the tape $T_t \in \mathbb{R}^{N_T \times d_T}$, the control state $Q_t \in \mathbb{R}^{d_Q}$, and the head index

$$I_t = (\boldsymbol{\theta}_t, \mathbf{w}_t) \in (S^1)^k \times \mathbb{R}^k$$

are learned, differentiable quantities. (Here $\boldsymbol{\theta}_t = (\theta_{t,1}, \dots, \theta_{t,k})$ parameterizes k head locations on the circle and $\mathbf{w}_t = (w_{t,1}, \dots, w_{t,k})$ are the associated weights.) We index tape positions by $j \in \{0, 1, \dots, N_T - 1\}$, and write $T_t[j] \in \mathbb{R}^{d_T}$ for the j-th tape symbol.

The initial state is produced by learned maps $\mathcal{M}_T, \mathcal{M}_Q, \mathcal{M}_I$ with parameters W:

$$T_0 = \mathcal{M}_T(x; W), \quad Q_0 = \mathcal{M}_Q(x; W), \quad I_0 = \mathcal{M}_I(x; W), \tag{2}$$

where $\mathcal{M}_T, \mathcal{M}_Q, \mathcal{M}_I$ can be any mapping using some parameters W.

3.2 Sparse addressing (keeping I and J fixed)

Define the following piecewise linear map $E : S^1 \to \mathbb{R}^{N_T}$ as

$$\begin{aligned}
n(\theta) &= \left\lfloor \frac{N_T \theta}{2\pi} \right\rfloor \\
s(\theta) &= \frac{N_T \theta}{2\pi} - \left\lfloor \frac{N_T \theta}{2\pi} \right\rfloor \\
n^+(\theta) &= (n(\theta) + 1) \bmod N_T \\
E(\theta) &= (1 - s(\theta)) e_{n(\theta)} + s(\theta) e_{n^+(\theta)}
\end{aligned}$$

We will write any $I \in (S^1)^k \times \mathbb{R}^k$ as $I = (\boldsymbol{\theta}, \mathbf{w})$, and define the induced sparse tape-index weighting vector $J(I) \in \mathbb{R}^{N_T}$ by

$$J(I) = \sum_{i=1}^{k} w_i \, E(\theta_i). \tag{3}$$

By construction, each $E(\theta_i)$ is supported on at most two adjacent tape locations $\{n(\theta_i), n^+(\theta_i)\}$, hence $J(I)$ is supported on at most $2k$ tape locations. Concretely, for each head atom (θ_i, w_i) define

$$n_i := n(\theta_i), \qquad s_i := s(\theta_i), \qquad n_i^+ := (n_i + 1) \bmod N_T,$$

so that $E(\theta_i) = (1 - s_i)e_{n_i} + s_i e_{n_i^+}$. This gives an implementation-friendly form: one can store $(n_i, n_i^+, (1 - s_i)w_i, s_i w_i)$ for each i and never materialize the dense N_T-vector $J(I)$.

3.3 Read, transition, and halting

We define the transition map Δ that updates the tape, control state, and head index. First, we use the head index $J(I_t)$ to read a single tape symbol $S_t \in \mathbb{R}^{d_T}$:

$$S_t = \sum_{j=0}^{N_T-1} \big(J(I_t)\big)_j \, T_t[j] \in \mathbb{R}^{d_T}. \tag{4}$$

Equivalently, using the explicit $2k$-sparse form above,

$$S_t = \sum_{i=1}^{k} w_{t,i} \Big((1 - s_{t,i}) \, T_t[n_{t,i}] + s_{t,i} \, T_t[n_{t,i}^+] \Big),$$

so S_t is computed using at most $2k$ gathered tape vectors, and is piecewise linear in the tape (and linear in the interpolation weights away from the measure-zero segment boundaries induced by the floor operation).

Next, define a gate $g_t \in (0, 1)$ that controls the effective amount of computation and enables early stopping. We use a sigmoid gate,

$$g_t = \sigma\left(\frac{-\kappa(x; W) \cdot t}{\max\big(1, \|Q_t - q_0\|^2\big)} \right), \tag{5}$$

where $\sigma(u) = 1/(1 + e^{-u})$, $\kappa(x; W) > 0$ is a learned scalar per example, and $q_0 \in \mathbb{R}^{d_Q}$ is a learned halting target state. Intuitively, $\kappa(x; W)$ acts as a *decay-rate multiplier*: smaller $\kappa(x; W)$ yields a slower decay in t (more effective steps), while larger $\kappa(x; W)$ yields a faster decay (fewer effective steps). The factor $\|Q_t - q_0\|$ encourages the dynamics to become stationary near the target.

We update the tape, control state, and head index using learned transition maps $\Delta_T, \Delta_Q, \Delta_\theta, \Delta_w$. Let

$$U_t := \Delta_T(S_t, Q_t; W) \in \mathbb{R}^{d_T}.$$

Then the update equations are

$$\begin{aligned}
T_{t+1}[j] &= T_t[j] + g_t \, \big(J(I_t)\big)_j \, U_t \qquad \text{for } j \in \{0, \dots, N_T - 1\}, &(6)\\
Q_{t+1} &= Q_t + g_t \, \Delta_Q(S_t, Q_t; W), &(7)\\
\boldsymbol{\theta}_{t+1} &= \big(\boldsymbol{\theta}_t + g_t \, \Delta_\theta(S_t, Q_t, \boldsymbol{\theta}_t; W)\big) \bmod 2\pi, &(8)\\
\mathbf{w}_{t+1} &= \mathbf{w}_t + g_t \, \Delta_w(S_t, Q_t, \mathbf{w}_t; W), &(9)\\
I_{t+1} &= (\boldsymbol{\theta}_{t+1}, \mathbf{w}_{t+1}). &(10)
\end{aligned}$$

Equation (6) makes the sparsity explicit: since $\big(J(I_t)\big)_j = 0$ for all but at most $2k$ locations, only $O(2k)$ tape vectors are updated per step. In an efficient implementation, (6) is executed as a scatter-add into those $2k$ indices (and S_t is computed as a gather + weighted sum).

The transition maps have the following types:

$$\begin{aligned}
\Delta_T &: \mathbb{R}^{d_T} \times \mathbb{R}^{d_Q} \to \mathbb{R}^{d_T},\\
\Delta_Q &: \mathbb{R}^{d_T} \times \mathbb{R}^{d_Q} \to \mathbb{R}^{d_Q},\\
\Delta_\theta &: \mathbb{R}^{d_T} \times \mathbb{R}^{d_Q} \times (S^1)^k \to \mathbb{R}^k,\\
\Delta_w &: \mathbb{R}^{d_T} \times \mathbb{R}^{d_Q} \times \mathbb{R}^k \to \mathbb{R}^k.
\end{aligned}$$

The $\mathrm{mod} 2\pi$ in (10) ensures the head angles represent elements of S^1 (equivalently, Δ_θ may be chosen 2π-periodic in each component). With sparse gather/scatter, one step costs $O(k(d_T + d_Q))$ time and $O(k(d_T + d_Q))$ working memory, plus the cost of evaluating the small transition networks.

Early stopping and block output. Fix a maximum unroll $T_{\max} \in \mathbb{N}$ and a threshold $\varepsilon > 0$. We run the transition until either $t = T_{\max}$ or the gate becomes negligible,

$$T(x) = \min\{t \in \{0, \dots, T_{\max} - 1\} : g_t < \varepsilon\},$$

with the convention $T(x) = T_{\max}$ if the set is empty. Concretely, we check $g_t < \varepsilon$ at the beginning of step t; if it holds, we stop and return T_t. Otherwise, we apply the transition to produce T_{t+1} and continue. We define the VECTUR block output as the final tape

$$V(x) := T_{T(x)} \in \mathbb{R}^{N_T \times d_T}.$$

In downstream architectures (e.g., Llama-style models), any required reshaping or projection of $V(x)$ is handled outside the VECTUR block.

Differentiability and efficient backpropagation. All operations inside each step are differentiable with respect to the tape values and the transition parameters, except at the measure-zero boundaries induced by the floor/mod operations inside $n(\theta)$. In practice, we implement reading and writing via gather/scatter on the at-most-$2k$ active indices, which is efficient and supports backpropagation through the unrolled computation. Early stopping introduces a discrete dependence on the stopping time $T(x)$; a standard choice is to stop the forward pass when $g_t < \varepsilon$ and treat the control-flow decision as non-differentiable, while gradients still flow through all executed steps (alternatively, one can always run for $T_{\max}$ steps and rely on the multiplicative g_t factors to effectively mask later updates).

Concrete parameterization (used in experiments). We instantiate the maps $\mathcal{M}_T, \mathcal{M}_Q, \mathcal{M}_I$ as linear projections, and the transition maps $\Delta_T, \Delta_Q, \Delta_w$ as two-layer MLPs with expansion factor 4. Specifically, we project tape symbols position-wise,

$$\mathcal{M}_T(x; W) = xW_T, \quad \mathcal{M}_T(z; W) = zW_T,$$

and define $\mathcal{M}_Q, \mathcal{M}_I$ as learnable linear maps that collapse the sequence to the required shapes. Writing

$$\operatorname{vec}(x) := [x[1]; x[2]; \cdots ; x[N]] \in \mathbb{R}^{N d_x},$$

we set

$$\mathcal{M}_Q(x; W) = W_Q \operatorname{vec}(x) \in \mathbb{R}^{d_Q}, \qquad \mathcal{M}_I(x; W) = \big(\boldsymbol{\theta}_0, \mathbf{w}_0\big),$$

with

$$\boldsymbol{\theta}_0 = (W_\theta \operatorname{vec}(x)) \bmod 2\pi \in (S^1)^k, \qquad \mathbf{w}_0 = W_w \operatorname{vec}(x) \in \mathbb{R}^k.$$

No constraint is imposed on $\mathbf{w}_0$; weights may be any real numbers.

We choose $\kappa(x; W)$ as a two-layer MLP (expansion factor 4) with a positivity constraint so that $\kappa(x; W) > 0$. For Δ_θ, we parameterize periodicity by feeding $\sin(\boldsymbol{\theta}_t)$ and $\cos(\boldsymbol{\theta}_t)$ into an MLP; concretely,

$$\Delta_\theta(S_t, Q_t, \boldsymbol{\theta}_t; W) = \mathrm{MLP}_\theta\big([S_t, Q_t, \sin(\boldsymbol{\theta}_t), \cos(\boldsymbol{\theta}_t)]\big) \in \mathbb{R}^k.$$

Algorithm (forward pass). Given (T_0, Q_0, I_0), we iterate for $t = 0, 1, \dots, T_{\max} - 1$:

1. compute $(n_{t,i}, s_{t,i}, n^+_{t,i})_{i=1}^k$ from $\boldsymbol{\theta}_t$ via the definitions above;
2. read S_t as a $2k$-term weighted sum of gathered tape vectors;
3. compute g_t; if $g_t < \varepsilon$, stop early and return T_t;
4. update Q_{t+1} and update $(\boldsymbol{\theta}_{t+1}, \mathbf{w}_{t+1})$;
5. write by scatter-adding into the at-most-$2k$ tape locations $\{n_{t,i}, n^+_{t,i}\}_{i=1}^k$ according to (6);

We return $V(x) = T_{T(x)}$.

4 VecTur Blocks inside Llama-style Models

4.1 Macro architecture

We adopt a standard decoder-only transformer macro architecture (token embeddings, positional encoding (Su et al., 2021), residual blocks, and an LM head) following Llama-family designs

Figure 2: **Placeholder.** Block-swap experiment: a fixed Llama-style macro architecture where the per-layer computational block is one of {Attention, LSTM, NTM/DNC, VecTur, VecSTur}.

(Touvron et al., 2023; Dubey et al., 2024). We then vary the *block* inside each residual layer while keeping parameter count and FLOPs roughly matched. This "block as inner loop" framing is inspired by recent work that integrates deliberate, multi-step computation into the forward pass via online learning or test-time adaptation, notably TTT-style test-time training (Tandon et al., 2025) and MIRAS-style online optimization views of sequence models (Behrouz et al., 2025). In that spirit, we view VecTur as an explicit, constrained "System 2" transition system embedded as a sub-layer inside a "System 1" decoder, rather than as a standalone memory system that replaces the macro architecture.

4.2 Compared blocks

We compare the following blocks:

- **Attention block**: multi-head self-attention (Vaswani et al., 2017) + SwiGLU MLP (Shazeer, 2020).
- **LSTM block**: a gated recurrent update applied over the sequence, wrapped with residual connections (Hochreiter and Schmidhuber, 1997).
- **External-memory block**: an NTM/DNC-style controller with differentiable read/write heads (Graves et al., 2014, 2016).
- **VecTur block**: the VecTur transition unrolled for $T_{\max}$ steps with learned halting κ.
- **VecSTur block**: VecTur with stochastic symbols z.

5 Evaluation Benchmarks

5.1 Reasoning and knowledge

We evaluate few-shot or fine-tuned performance on:

- **GSM8K** (Cobbe et al., 2021) (grade-school math; exact-match accuracy),
- **ARC** (Clark et al., 2018) (AI2 reasoning challenge; accuracy),
- **HellaSwag** (Zellers et al., 2019) (commonsense completion; accuracy).

5.2 Language modeling

We evaluate next-token prediction on **WikiText-103** (Merity et al., 2016) using perplexity.

6 CompGen: Complexity-Stratified Program Generation

6.1 Task format

CompGen consists of short Python programs p paired with inputs u and outputs $p(u)$. Each instance is labeled with a target complexity class $(T(n), S(n))$ in terms of input size n (Sipser, 2012). Programs are generated from templates with controlled loop structure, recursion depth, and memory allocation patterns. We view CompGen as a utility dataset in the tradition of synthetic algorithmic benchmarks, complementary to the CLRS Algorithmic Reasoning Benchmark (Veličković et al., 2022).

Class	Example family	Notes
$O(n), O(1)$	scan / reduce	single pass
$O(n), O(n)$	prefix sums	linear auxiliary array
$O(n \log n), O(1)$	sort-then-scan	comparison sorting
$O(n^2), O(1)$	nested-loop count	quadratic time
$O(n^2), O(n)$	DP table strip	quadratic time, linear space

Table 1: **Placeholder.** COMPGEN program families and intended $(T(n), S(n))$ buckets.

6.2 Generalization protocol

We train on $n \in [n_{\min}, n_{\text{train}}]$ and evaluate on larger $n \in (n_{\text{train}}, n_{\text{test}}]$ to measure extrapolation. We report accuracy as a function of n and correlate effective compute (average unroll steps) with complexity class.

7 Randomized Computation Suite

We include tasks where access to randomness enables provable or empirical speedups:

- **Matrix product verification** (Freivalds) (Freivalds, 1977): verify $AB = C$ faster than multiplication.
- **Probabilistic primality testing** (Miller–Rabin) (Miller, 1976; Rabin, 1980): decide primality with bounded error.

VECSTUR receives stochastic symbols z and learns to leverage them to reduce expected compute (as reflected by learned κ and early halting).

8 Experimental Setup

Model sizes. We instantiate models at $\sim$110M, 350M, and 1.3B parameters (placeholder sizes) with matched embedding width and layer count across blocks.

Blocks and controlled comparisons. Unless otherwise stated, we run the same experiment for each block in Section 3 (Attention, LSTM, NTM/DNC, VECTUR, VECSTUR), holding the decoder-only macro architecture fixed and matching parameter count and training budget as closely as possible.

Experimental protocols (run per block). We use three complementary training/evaluation protocols:

1. **Language pretraining $\rightarrow$ downstream evaluation.** We pretrain on **FineWeb** (general web text), then evaluate on **GSM8K** (Cobbe et al., 2021), **ARC** (Clark et al., 2018), **HellaSwag** (Zellers et al., 2019), and **WikiText-103** (Merity et al., 2016). (Table 2.)
2. **Algorithmic transfer between CLRS and COMPGEN.** (a) **Train on CLRS** (Veličković et al., 2022) and evaluate on COMPGEN under three regimes: *zero-shot* (no COMPGEN training), *few-shot* (in-context demonstrations at test time), and *fine-tune* (supervised adaptation on COMPGEN train). (b) **Train on COMPGEN** and evaluate on a held-out COMPGEN split (including out-of-distribution generalization across input sizes n per Section 5.2). (Figure 3.)
3. **In-domain CLRS generalization.** We train on CLRS (Veličković et al., 2022) and evaluate on a held-out CLRS split (standard in-distribution generalization across graphs/sizes/instances). (Reported alongside other algorithmic results; placeholder in this draft.)

Optimization and budgets. Within each protocol, we use identical optimizers, learning rate schedules, and token/step budgets across blocks (to isolate architectural effects).

Compute control. For VECTUR/VECSTUR we set a maximum unroll $T_{\max}$ and learn $\kappa(x; W)$ to modulate effective steps. We report both task performance and measured compute (average unroll steps per token).

Block (model)	Train set	GSM8K (test)	ARC (test)	HellaSwag (test)	WikiText-103 (test)
Attention	FineWeb	Lorem	Ipsum	Dolor	Sit
LSTM	FineWeb	Amet	Consectetur	Adipiscing	Elit
NTM/DNC	FineWeb	Sed	Do	Eiusmod	Tempor
VECTUR	FineWeb	Incididunt	Ut	Labore	Et
VECSTUR	FineWeb	Magna	Aliqua	Ut	Enim

Table 2: **Placeholder (Protocol 1).** Language pretraining on FineWeb, evaluated on downstream benchmarks. Entries are Lorem ipsum placeholders.

Block	Train	Test	Result
Attention	CLRS	COMPGEN (zero-shot)	Lorem ipsum
Attention	CLRS	COMPGEN (few-shot)	Dolor sit
Attention	CLRS	COMPGEN (fine-tune)	Amet consectetur
LSTM	CLRS	COMPGEN (zero-shot)	Adipiscing elit
LSTM	CLRS	COMPGEN (few-shot)	Sed do
LSTM	CLRS	COMPGEN (fine-tune)	Eiusmod tempor
NTM/DNC	CLRS	COMPGEN (zero-shot)	Incididunt ut
NTM/DNC	CLRS	COMPGEN (few-shot)	Labore et
NTM/DNC	CLRS	COMPGEN (fine-tune)	Magna aliqua
VECTUR	CLRS	COMPGEN (zero-shot)	Ut enim
VECTUR	CLRS	COMPGEN (few-shot)	Ad minim
VECTUR	CLRS	COMPGEN (fine-tune)	Veniam quis
VECSTUR	CLRS	COMPGEN (zero-shot)	Nostrud exercitation
VECSTUR	CLRS	COMPGEN (few-shot)	Ullamco laboris
VECSTUR	CLRS	COMPGEN (fine-tune)	Nisi ut

Table 3: **Placeholder (Protocol 2a).** Train on CLRS, test on COMPGEN under zero-shot / few-shot / fine-tune adaptation regimes. Results are placeholders.

9 Results (Illustrative Placeholders)

Important note. The tables below contain **Lorem ipsum placeholder entries** showing the intended presentation format *and* explicitly recording the train/test split for each experiment protocol. Replace these placeholders with measured metrics.

10 Discussion

These illustrative results suggest VECTUR provides a useful inductive bias for tasks requiring iterative computation and length extrapolation, while remaining compatible with modern LLM macro architectures. Importantly, the strongest claims in this paper are *not* that differentiable Turing machines are new, but that (i) enforcing strictly local sparse access yields a practical, non-blurring pointer-machine-style block, (ii) treating such a machine as a composable Transformer sub-layer is a strong systems contribution, and (iii) VECSTUR highlights a comparatively underexplored angle: learning to exploit randomness as a computational resource in randomized-algorithm tasks. VECSTUR further improves performance on tasks where randomized strategies are advantageous.

11 Limitations and Future Work

This draft omits implementation details (e.g., the exact $\mathrm{Sparse}(\cdot)$ operator, stability constraints, and efficient kernels) and uses illustrative results. Future work should (i) benchmark on longer-context settings, (ii) analyze failure modes of learned halting κ, and (iii) evaluate robustness across different data mixtures and training budgets.

Block	Train	Test	Result
Attention	CompGen (train)	CompGen (held-out)	Lorem ipsum
LSTM	CompGen (train)	CompGen (held-out)	Dolor sit
NTM/DNC	CompGen (train)	CompGen (held-out)	Amet consectetur
VecTur	CompGen (train)	CompGen (held-out)	Adipiscing elit
VecSTur	CompGen (train)	CompGen (held-out)	Sed do

Table 4: **Placeholder (Protocol 2b).** Train on CompGen, test on held-out CompGen (including extrapolation across larger n). Results are placeholders.

Block	Train	Test	Result
Attention	CLRS (train)	CLRS (held-out)	Lorem ipsum
LSTM	CLRS (train)	CLRS (held-out)	Dolor sit
NTM/DNC	CLRS (train)	CLRS (held-out)	Amet consectetur
VecTur	CLRS (train)	CLRS (held-out)	Adipiscing elit
VecSTur	CLRS (train)	CLRS (held-out)	Sed do

Table 5: **Placeholder (Protocol 3).** Train on CLRS and evaluate on a held-out CLRS split. Results are placeholders.

11.1 Future Work: Mechanistic Interpretability

VecTur is unusually well-suited for mechanistic interpretability (Olah et al., 2020; Elhage et al., 2021) because its learned dynamics are constrained to resemble an explicit Turing-style transition system: a finite-dimensional control state Q_t, a tape T_t, and a small number of heads with sparse, local read/write effects. This structure encourages explanations in terms of *state machines* and *pointer-based algorithms* (e.g., "scan until condition," "increment counter," "copy span," "simulate update rule"), rather than opaque global attention patterns.

A promising direction is to *disassemble* trained VecTur blocks into more directly inspectable artifacts. For example, one can post-hoc discretize head locations, identify stable control states, and summarize the transition maps Δ as a symbolic program or a finite set of guarded update rules; such representations can then be *transpiled* into executable code, enabling unit tests, counterfactual interventions, and formal analysis of the implied algorithm.

Finally, VecTur may serve as an interpretable *surrogate* for black-box sequence models. Analogous to knowledge distillation (Hinton et al., 2015; Romero et al., 2015), one can perform *cross-distillation*: train a VecTur model to mimic the input–output behavior (and, when available, internal activations) of an existing architecture, with the goal that the learned tape-and-control dynamics provide a concrete hypothesis for the black box's implicit Turing-style computation. Such surrogates could support "algorithmic guessing"—extracting candidate programs from the VecTur dynamics—followed by validation against the teacher via targeted probes and adversarial test cases.

Acknowledgments

Placeholder.

References

Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 1936.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 1997.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NeurIPS*, 2017.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural Turing machines. *arXiv:1410.5401*, 2014.

Figure 3: **Placeholder.** CompGen extrapolation: accuracy vs. input size n, showing how blocks degrade with larger n and how VecTur modulates effective steps via learned κ.

Block	Train set	Test set	Result
VecTur	Randomized suite (train)	Freivalds / Miller–Rabin (test)	Lorem ipsum
VecSTur	Randomized suite (train)	Freivalds / Miller–Rabin (test)	Dolor sit amet

Table 6: **Placeholder.** Randomized computation suite: train/test bookkeeping with placeholder results.

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 2016.

Alex Graves. Adaptive computation time for recurrent neural networks. *arXiv:1603.08983*, 2016.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv:2307.09288*, 2023.

Abhimanyu Dubey et al. The Llama 3 herd of models. *arXiv preprint*, 2024.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukas Kaiser, Matthias Plappert, et al. Training verifiers to solve math word problems. *arXiv:2110.14168*, 2021.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try ARC, the AI2 reasoning challenge. *arXiv:1803.05457*, 2018.

Rowan Zellers, Yonatan Bisk, Roy Schwartz, and Yejin Choi. HellaSwag: Can a machine really finish your sentence? In *ACL*, 2019.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv:1609.07843*, 2016. (Introduces the WikiText-103 benchmark.)

Raimund Freivalds. Probabilistic machines can use less running time. In *IFIP Congress*, 1977.

Gary L. Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences*, 1976.

Michael O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 1980.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, et al. Language models are few-shot learners. In *NeurIPS*, 2020.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers. In *ICLR*, 2019.

Andrea Banino, Jan Balaguer, Charles Blundell, and Andrew Zisserman. PonderNet: Learning to ponder. In *NeurIPS*, 2021.

Ali Behrouz, Meisam Razaviyayn, Peilin Zhong, and Vahab Mirrokni. It's All Connected: A Journey Through Test-Time Memorization, Attentional Bias, Retention, and Online Optimization. *arXiv:2504.13173*, 2025.

Nelson Elhage, Sam S. McCandlish, Catherine Olsson, Christopher Henighan, Nicholas Joseph, Ben Mann, Seth Kaplan, et al. A mathematical framework for transformer circuits. *Transformer Circuits (Anthropic)*, 2021.

Łukasz Kaiser and Ilya Sutskever. Neural GPUs learn algorithms. In *ICLR*, 2016.

Chris Olah, Nick Cammarata, Ludwig Schubert, Gabriel Goh, Michael Petrov, and the OpenAI Clarity team. Zoom in: An introduction to circuits. *Distill*, 2020.

Ofir Press, Noah A. Smith, and Mike Lewis. Train short, test long: Attention with linear biases enables input length extrapolation. In *ICLR*, 2022.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv:1503.02531*, 2015.

Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. FitNets: Hints for thin deep nets. In *ICLR*, 2015.

Noam Shazeer. GLU variants improve transformer. *arXiv:2002.05202*, 2020.

Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 3rd edition, 2012.

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

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *NeurIPS*, 2015.

Petar Veličković, Adrià Puigdomènech Badia, David Budden, Razvan Pascanu, Andrea Banino, Misha Daswani, Raia Hadsell, and Charles Blundell. The CLRS Algorithmic Reasoning Benchmark. *arXiv preprint arXiv:2205.15659*, 2022.

Arnuv Tandon, Karan Dalal, Xinhao Li, Daniel Koceja, Marcel Rød, Sam Buchanan, Xiaolong Wang, Jure Leskovec, Sanmi Koyejo, Tatsunori Hashimoto, Carlos Guestrin, Jed McCaleb, Yejin Choi, and Yu Sun. End-to-End Test-Time Training for Long Context. *arXiv:2512.23675*, 2025.