VecTur: Vector Turing Machines

Ethan Hall
ethan.hall.phd@gmail.com

Abstract

We introduce VECTUR (Vector Turing Machines), a differentiable architecture for learning functions by vectorizing the tape symbols, head index, and finite control of a classical Turing machine (Turing, 1936). VECTUR is designed to support *deep computation* by explicitly iterating a learned transition map while learning an input-conditioned halting schedule via a parameter κ. 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 at fixed parameter budgets. We additionally introduce COMPGEN—a synthetic program dataset stratified by $(T(n), S(n))$ complexity classes—and show VECTUR adapts its effective compute by learning κ. Finally, we propose VECSTUR, a stochastic extension that consumes random tape symbols and 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, or allocate variable compute per input. 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. Unlike attention-only computation (Vaswani et al., 2017), VECTUR explicitly iterates a state transition; unlike Neural Turing Machines (Graves et al., 2014), VECTUR emphasizes sparse head indexing and a learned halting schedule parameterized by κ, enabling adaptive depth.

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.

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).

Contributions.

- We define VecTur, a vectorized Turing-machine-inspired transition system with sparse indexing and a learnable halting schedule κ.
- We propose a plug-and-play integration of VecTur as a computational block inside Llama-style decoder-only models.
- We introduce CompGen, a synthetic program dataset labeled by time/space complexity class $(T(n), S(n))$, and a protocol for extrapolation across n.
- We define VecSTur and a randomized computation evaluation suite, showing benefits of stochastic symbols for problems with randomized speedups.

2 Related Work

Sequence models and attention. Transformers (Vaswani et al., 2017) and their decoder-only variants power modern LLMs (e.g., 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 (i) sparse indexing for efficiency and (ii) a learned halting schedule.

Adaptive computation. Adaptive Computation Time (ACT) (Graves, 2016) learns when to stop iterating; our κ-parameterization provides a simple, input-conditioned control knob for the effective number of steps.

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 $\boldsymbol{\theta}_t = (\theta_{t,1}, \ldots, \theta_{t,k})$ parameterizes k head locations on the circle and $\mathbf{w}_t = (w_{t,1}, \ldots, w_{t,k})$ are the associated weights.) We index tape positions by $j \in \{0, 1, \ldots, N_T - 1\}$, 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,

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

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, larger $\kappa(x;W)$ yields more effective steps (slower decay in t), while $\|Q_t - q_0\|$ 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

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

Equation (6) makes the sparsity explicit: since $\left(J(I_t)\right)_j = 0$ 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

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

we set

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

Figure 1: **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}.

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 $t = 0, 1, \ldots, T_{\max} - 1$:

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, residual blocks, and an LM head) following Llama-family designs (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.

4.2 Compared blocks

We compare the following blocks:

- **Attention block**: multi-head self-attention + SwiGLU MLP (Vaswani et al., 2017).
- **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:

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.

- **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. Programs are generated from templates with controlled loop structure, recursion depth, and memory allocation patterns.

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.

Training. We train on a mixture of general text (for language modeling) and CompGen (for compute probing), then fine-tune on downstream reasoning tasks. We use identical optimizers, learning rate schedules, and token budgets across conditions.

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	**GSM8K**	**ARC**	**HellaSwag**	**WikiText PPL**
Attention	42.1	54.0	78.3	18.7
LSTM	38.4	51.2	76.0	20.9
NTM/DNC	43.0	54.4	78.1	19.3
VecTur	**46.8**	**56.1**	**79.0**	18.9
VecSTur	46.2	55.7	78.9	18.9

Table 2: **Placeholder.** Benchmark performance for a mid-size model at matched parameter budget. Accuracy in %, perplexity lower is better.

Figure 2: **Placeholder.** CompGen extrapolation: accuracy vs. input size n, showing VecTur degrades more gracefully and increases effective steps via learned κ.

9 Results (Illustrative Placeholders)

Important note. The tables below contain **illustrative placeholder numbers** showing the intended presentation format. Replace these with actual experimental results.

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. 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.

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.

Model	**Freivalds verify**	**Miller–Rabin**
VecTur	71.0	68.4
VecSTur	**84.5**	**79.2**

Table 3: **Placeholder.** Randomized computation suite: VecSTur benefits from stochastic symbols z.

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

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.