Universal and Efficient Computation with 2D Attention

Christos Tzamos **Guoqing Zheng** **Athul Paul Jacob**
Percepta
{ctzamos, gzheng, apjacob}@percepta.ai

Abstract

Transformers are powerful models but are inefficient with large context due to the quadratic complexity of the attention mechanism. Limiting their context window addresses the efficiency concern but at a cost: they cannot remember everything from the past, which severely limits their computational power. We show that we can simultaneously achieve efficiency and universal computation if we restrict attention heads to be 2-dimensional. First, we give the first algorithm for standard 2D attention that serves every insertion and every query in amortized polylogarithmic time per token. Prior work either restricted key/query magnitudes or paid polynomial overhead per step. Second, we show that the class of 2D-attention models is universal and efficient: a vanilla 2D transformer efficiently simulates an arbitrary RAM machine via chain-of-thought decoding so that, combined with our first result, T steps of a word-RAM machine can be simulated in $\widetilde{O}(T)$ time. In contrast, earlier universality results focused on computability rather than per-step efficient simulation. Third, as an application, we turn a transformer into a computer. We provide a compiler that supports the full `i32` WebAssembly subset with integer computations (reachable from C via `clang`) and turns such programs into the weights of a transformer, which then executes them over millions of steps autoregressively. We illustrate how this can be used to solve combinatorial problems, tasks at which even large reasoning LLMs struggle.

1 Introduction

Modern transformers owe much of their power to the attention mechanism, which at every step lets the next token query the entire prior context. This flexibility comes at a cost: generating a length-N sequence with standard attention takes $\Theta(N^2)$ time, so deterministic computations that span millions of tokens are prohibitively expensive in the autoregressive regime that current language models use. Existing efficient alternatives reach long contexts by trading expressivity for speed: windowed attention (Xiao et al., 2024) caps the number of past tokens visible to each query, while state-space models (Gu and Dao, 2024) and linear attention (Katharopoulos et al., 2020) replace the per-token attention sum with a fixed-size summary of the past. The main motivation behind our work is whether this tradeoff between expressivity and speed is inherently required.

Our main result is that a subclass of transformers, namely 2D-attention transformers, are **both efficient and universal**. In fact, arbitrary computation in a RAM machine can be simulated efficiently by a transformer with polylogarithmic overhead.

A standard transformer of model width d_{model} splits each layer into $H = d_{\text{model}}/d_h$ attention heads, where every head computes attention through inner products of queries and keys in $\mathbb{R}^{d_h}$. We show that if we fix the per-head dimension to $d_h = 2$, the resulting class of *2D-attention* models admits a polylogarithmic-per-token attention algorithm (Result 1) and is universal in the sense of constructive Turing completeness (Result 2). This restriction is only on the per-head dimension; the model width

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

and depth remain unconstrained, so a 2D-head model can be scaled to billions of parameters in the same way as a standard transformer (Shazeer, 2019; Ainslie et al., 2023).

Two dimensions are the smallest non-trivial setting that allows focusing on a single position/token from the past. Higher dimensions become exponentially more expensive to maintain online, with conditional lower bounds ruling out polynomial speedups beyond logarithmic dimension (§2). Exploiting the planar geometric structure, our first result shows that attention can be computed exponentially faster than the general case while our second result shows that the class of 2D-models are expressive enough to perform arbitrary computation:

Result 1: An exponentially faster algorithm for 2D attention. We give the first algorithm for 2D attention that serves every insertion and every query in amortized polylogarithmic time per token. This exponentially improves the standard quadratic attention that requires linear time per step. Our algorithm handles keys and queries of arbitrary norms and works in the challenging online setting of autoregressive generation, where queries, keys, and values arrive one at a time. Prior subquadratic attention algorithms either restricted query and key magnitudes (Alman and Song, 2023) or paid polynomial overhead per token (Gupta et al., 2026).

Result 2: 2D-transformers are universal computers. We show that 2D transformers are expressive enough to perform arbitrary computation. Concretely, we construct a single fixed 2D transformer that simulates any Turing machine M on any input x when the encoded pair $\langle M, x\rangle$ is supplied as the prefix; under unbounded integer precision this is infinite-horizon exact Turing completeness, and simulating T steps of a Turing machine on an input of length n requires only $O(\log(n+T))$ bits of precision. This extends the work of Pérez et al. (2021), which used a separate input encoder for Turing completeness and relied on a more complex attention scoring rule. We also show that an $O(\log n)$-bit word-RAM machine on inputs of size n can be simulated efficiently step by step. Over T simulated steps, the transformer takes $\mathrm{polylog}(n+T)$ amortized time per RAM step (Corollary 3.6). Earlier universality results either rely on architectures with a separate input encoder, non-standard scoring, or unbounded precision (Pérez et al., 2021; Bhattamishra et al., 2020), or characterize complexity classes without giving per-step efficient simulation (Merrill and Sabharwal, 2024); §6 surveys this line.

Application: Running code inside a transformer. As an application of our previous efficient construction, we provide a compiler that supports the full `i32` WebAssembly subset with integer computations (reachable from C via `clang`) and turns such programs into the weights of a 2D transformer. We illustrate this by compiling complex solvers written in C for combinatorial tasks (like Sudoku) to the weights of the model. The model then reads the input and runs the code autoregressively, producing execution tokens until it arrives at a solution. The compiler and examples are available online.[1] Compilation of logic into transformer weights has direct precedents in Tracr (Lindner et al., 2023) and Looped Transformers (Giannou et al., 2023); we extend this line by targeting C-reachable integer control flow and a standard autoregressive 2D transformer, with the detailed comparison in §6. As a sanity check for the compiler, we compile a deterministic backtracking Sudoku solver, which demonstrates exact long-horizon symbolic execution inside the constructed 2D transformer; this is not meant as a learned-Sudoku benchmark comparison, though we note that existing reasoning LLMs reach near 0% accuracy on hard instances without external tools (Giannoulis et al., 2025).

In the following sections, we describe our results in more detail.

2 Fast 2D Attention

In this section, we focus on a single **2D-attention head** in isolation. The attention head works as follows: At step i, after receiving a (key, value) pair $(k_i, v_i) \in \mathbb{R}^2 \times [-V, V]^d$ with value dimension $d = O(1)$ and a query $q_i \in \mathbb{R}^2$, the head must return the sum of all past values, weighted by the softmax of the inner-products between the query and the past keys. In particular, we must compute $A_i = Y_i/Z_i$ where $Y_i = \sum_{j \le i} v_j \, e^{\langle q_i, k_j\rangle}$ and $Z_i = \sum_{j \le i} e^{\langle q_i, k_j\rangle}$.

[1] https://github.com/Percepta-Core/transformer-vm

Our main result is an efficient approximate 2D-attention algorithm which for any $\varepsilon > 0$ computes $\hat{A}_i$ such that $\|\hat{A}_i - A_i\|_\infty \le \varepsilon$. Our algorithm targets the *online* setting required by inference, where queries, keys, and values arrive one at a time and must be answered immediately. This is the most general case, and the static batch setting follows as a special case.

Theorem 2.1 (Efficient Online 2D-Attention; proof: Appendix A)**.** *For two-dimensional keys and queries (of arbitrary magnitude), there is an online deterministic data structure that supports insertions of key, value pairs and ε-approximate 2D-attention queries. The amortized time of either operation over a sequence of N operations is* $\mathrm{polylog}(NV/\varepsilon)$.

We note that our theorem can approximate arbitrary dense softmax attention in 2-dimensions. In contrast, methods like sparse attention impose a specific sparsity pattern and can be very lossy for heads depending on many past values.

Table 1: Comparison with prior work for approximate 2D attention with N tokens

	Total time	Norms	Technique
Alman and Song (2023)	$\widetilde{O}(N)$	bounded	Taylor Moments
Gupta et al. (2026)	$\widetilde{O}(N^{3/2})$	generic	Taylor Moments + Localization
This work (online)	$\boldsymbol{\widetilde{O}(N)}$	**generic**	Taylor Moments + **Efficient** Localization

Prior to our work, Alman and Song (2023) obtain near-linear softmax attention $N^{1+o(1)}$ but only when both query and key norms are bounded by $O(\sqrt{\log N})$. In that regime they represent the kernel by its degree-$O(\log N)$ moments, reducing each row to a polynomial sum of $O(\log^d N)$ dimensions ($O(\log^2 N)$ at $d = 2$). While the bounded norm restriction may not seem significant, it effectively reduces the problem to a fixed-size linear-attention summary (Katharopoulos et al., 2020) of the past whose representational capacity is only polylogarithmic in the prefix length. Such algorithms therefore replace the prefix by a polylog-dimensional moment summary, which is insufficient for the exact addressable-memory behavior used in our universality construction.

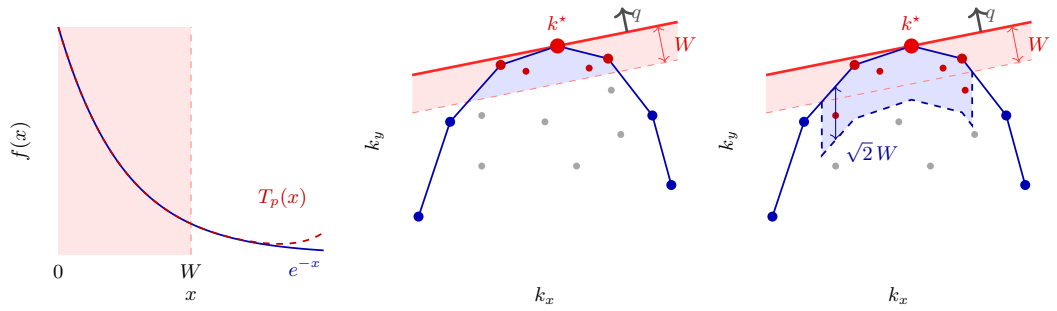
Gupta et al. (2026) address these limitations: their algorithm works for arbitrary norms and preserves the representational capacity of full attention by localizing the Taylor expansion to an active region around the maximum at a cost of polynomial overhead per query: $\widetilde{O}(N^{1-1/d})$ (i.e. $\widetilde{O}(N^{1/2})$ at $d = 2$).

Our algorithm uses the same localized Taylor expansion as Gupta et al. (2026) but replaces their halfspace range search by a 2D rectangular range search via a range tree, yielding polylogarithmic time per token in the online setting at any norm. We next describe our construction in more detail.

Proof sketch. We first describe the construction of a static data structure that answers queries against a fixed key set in polylog time, and then make it online via a **doubling trick**.

Approach. We write the softmax output as $A_i = Y_i/Z_i$ and approximate the numerator Y_i and denominator Z_i separately. Both are exponential sums, so following Alman and Song (2023) we approximate the exponential by a degree-p Taylor polynomial and compute each sum by maintaining p^2 bivariate moments, but this works only if keys and queries are bounded. We localize the Taylor expansion to the subset of keys that contribute non-negligibly to the exponential sum, an idea due to Gupta et al. (2026): given a query q against the key set K, we let $m_q = \max_{k\in K}\langle q, k\rangle$ denote the maximum score and define the *active set* $\{k : \langle q, k\rangle \ge m_q - W\}$ for a width $W = \Theta(\log(NV/\varepsilon))$. A degree-$\Theta(W)$ Taylor polynomial of e^{-x} then approximates the kernel uniformly on the active set, while the remaining keys jointly contribute at most $\varepsilon Z_i/(8V)$ to the denominator and $\varepsilon Z_i/8$ in absolute value to each numerator coordinate (Figure 1(a)), and can be discarded.

From half-plane to rectangle. The active set is the intersection of K with a half-plane (Figure 1(b)). Gupta et al. (2026) answer this half-plane query directly via the simplex partition tree of Matoušek (1992) at $\widetilde{O}(\sqrt{N})$ per query, which is the source of their $\widetilde{O}(N^{3/2})$ total time-complexity. We avoid the half-plane query altogether by exploiting the fact that enlarging the active set to any superset contained in the wider band $\langle q, k\rangle \ge m_q - 2W$ still keeps the error within ε. Concretely, we partition queries into four sectors by dominant direction (up, down, left, or right) and keep one data structure per sector. WLOG consider the upward sector $q_y \ge |q_x|$, and let $h(\cdot)$ denote the concave



(a) Taylor approximation T_p of e^{-x} (b) Halfspace range search (Gupta et al., 2026) (c) Rectangle range search (this work)

Figure 1: Geometric backbone of 2D attention. The softmax kernel needs an active region around the maximum, viewed three ways below. **(a)** The kernel e^{-x} has a degree-p Taylor surrogate T_p uniformly accurate on $[0, W]$ (with $p = \Theta(W + \log(1/\varepsilon))$); outside $[0, W]$ the kernel itself is exponentially small. **(b)** Gupta et al. (2026) use the halfspace $\{k : \langle q, k\rangle \geq m_q - W\}$ with $W = \Theta(\log(NV/\varepsilon))$ as the active region (the parallelogram strip below the support line at $k^\star$), at $\widetilde{O}(\sqrt{N})$ per query. **(c)** *Our work:* we replace the halfspace by an axis-aligned rectangle in shifted coordinates obtained by subtracting the upper hull from the dominant axis (up, down, left, right). Every red point has score gap to the halfspace at most $2W$ and the rectangle is a controlled superset of the active set. The 2D rectangular range query can be answered in $\mathrm{polylog}(N)$ (Theorem 2.1).

piecewise-linear function whose graph is the upper hull of the keys. We map every key $k = (k_x, k_y)$ to $k' = (k_x, k_y - h(k_x))$, flattening the hull to the x-axis (Figure 1(c)); every shifted key now has $k'_y \leq 0$, with equality on the hull. In shifted coordinates there is always an axis-aligned rectangle whose intersection K_R with K satisfies $\{k : \langle q, k\rangle \geq m_q - W\} \subseteq K_R \subseteq \{k : \langle q, k\rangle \geq m_q - 2W\}$.

Rectangle range search. Answering a query q over K_R is then straightforward: binary search on the upper hull locates the interval $[k^{\mathrm{lo}}, k^{\mathrm{hi}}]$ and its height in the y direction is determined by q. A standard 2D rectangular range query returns this rectangle as a disjoint union of $O(\log^2 n)$ canonical subsets in $O(\log^2 n)$ time, independent of the rectangle's dimensions or the keys' magnitudes. At each canonical subset we precompute bivariate Taylor moments of the original keys; the kernel sum then reduces to a contraction of these moments against the Taylor coefficients of e^{-x}.

Online via doubling. We turn this static data structure into a dynamic one by a doubling trick. At any prefix length t we maintain $O(\log N)$ static structures aligned with the binary expansion of t, structure ℓ covering a contiguous block of 2^ℓ recently inserted keys. Insertion induces the binary-counter merge pattern, and standard amortization keeps the per-insertion cost polylogarithmic. A query runs the static query against each live structure and sums the contributions to the numerator and denominator; the per-query cost remains polylogarithmic. □

Beyond two dimensions. Our algorithm exploits planar geometry: a 1D slope variable along the upper hull parameterizes queries, and a 2D rectangular range search returns the active region in polylog time. Higher dimensions are much more expensive: the Taylor expansion requires $\Theta(W^d)$ moments per canonical node for fixed d, and the conditional SETH lower bound of Alman and Song (2023); Duman Keles et al. (2023); Gupta et al. (2026) rules out any polynomial improvement over N^2 at $d = \omega(\log N)$ (only sub-polynomial improvements such as $N^2/\log N$ remain possible). Two dimensions sit at a balanced point: enough expressivity to support universal computation (§3) and enough planar structure to admit a polylog-per-token algorithm. This motivates the 2D-head architectural choice used throughout the rest of the paper.

3 Universal Computation by 2D Transformers

Section 2 showed that 2D attention can be computed efficiently. We now show that it is also expressive enough for arbitrary computation: we introduce an abstract computation model, the *Append-only Lookup Machine* (ALM), that 2D transformers implement exactly and that is itself Turing complete. Section 3.1 defines the ALM and proves that every ALM compiles into a 2D transformer of constant

depth and width. Section 3.2 proves that an ALM with only four memories simulates an arbitrary Turing machine, so the 2D transformer class itself is Turing complete.

3.1 Transformers as Append-only Lookup Machines

We define the ALM as an autoregressive per-token computation model. At every position the machine emits the next token through a fixed schedule of *write–read–compute* cycles against a constant number of integer-indexed memories that grow with the prefix. We restrict reads to only two primitives, point lookup and cumulative sum, and we control keys, queries, and the emitted token through a constant-depth combinational circuit between memories. We choose this primitive set precisely because each primitive maps onto a constant number of transformer-native operations; this is what lets us compile any ALM exactly into a 2D transformer of constant depth and width.

Definition 3.1 (Append-only Lookup Machine)**.** An ALM has a finite alphabet Σ and a constant number $r \in \mathbb{N}$ of *memories* $\mathcal{M}_1, \ldots, \mathcal{M}_r$. Each memory $\mathcal{M}_j$ is a plain integer-indexed array $\mathcal{M}_j : \mathbb{Z} \to \mathbb{Z}^{c_j}$ of constant value width c_j, initially zero everywhere.

The ALM is autoregressive: it generates a token sequence $w_0, w_1, w_2, \ldots$ by recomputing the next token at every position from the prefix, through a single sweep through the memories in fixed order $1, 2, \ldots, r$. For each memory $\mathcal{M}$ visited in turn, the step performs the following *write–read–compute* cycle:

1. **Write.** Pick an integer key k and a value v, and set $\mathcal{M}[k] := v$. Any value previously stored at index k is overwritten.
2. **Read.** Perform a constant number of reads on $\mathcal{M}$. Each read is one of two primitives:
   - **Lookup** at a query q: return $\mathcal{M}[q]$ (the current value at index q, or zero if nothing has been written at q).
   - **Cumulative sum**: return the sum of all values that have been written into $\mathcal{M}$ at this and earlier positions, including values that were later overwritten.
3. **Compute.** A constant-depth, constant-width combinational circuit over the values read so far determines the keys, values, and queries used at the next memory. After the rth memory, the same kind of circuit computes and returns the next token $w_{i+1} \in \Sigma$.

With the model fixed, we now make the correspondence to transformers precise: every ALM compiles exactly into a 2D transformer of constant depth and width, with each ALM primitive realized by a constant number of transformer-native operations.

Theorem 3.2 (2D transformers implement ALMs; proof: Appendix B.3)**.** *For every ALM A we construct a causal 2D transformer T_A of constant depth and width, both determined by A, that emits the same token sequence as A on every input prefix.*

Proof sketch. The structural form of the ALM mirrors the structural form of an autoregressive 2D transformer layer: a fixed schedule of memory accesses interleaved with constant-depth local computation, repeated at every position. We exploit this correspondence directly: we realize each scalar ALM read, whether a lookup or a cumulative sum, with one attention head, and we use the residual stream to carry the values of every memory and gate.

At each position we use one attention head per scalar ALM read: the head's K and V projections at the current token write the memory's chosen key k and value v, and the head's Q projection issues the read query. A memory with multiple reads uses one head per scalar read. We realize the two read primitives with two distinct attention patterns:

- For *lookup*, we use the parabolic key encoding $k \mapsto (2k, -k^2)$. Under hardmax attention with query $(q, 1)$, the score $2qk - k^2 = -(k-q)^2 + q^2$ is uniquely maximized at $k = q$ among integer keys. A positional tie-break selects the latest position whose stored key equals q (Lemma B.5).
- For *cumulative sum*, we use uniform attention ($W_Q = W_K = 0$): every logit is identical, so attention returns the prefix average $\frac{1}{i+1} \sum_{t \le i} v(t)$, and a downstream gate that multiplies by $\mathbf{pos} + \mathbf{1}$ recovers the exact prefix sum.

We realize the constant-depth compute step between memories by a constant number of feed-forward neurons; on integer inputs, the gating pattern $a \cdot \mathrm{ReLU}(b)$ derives step indicators, equality, conditional selection, and integer multiplication via standard ReLU identities. We give the full construction, including the column layout of the residual stream and explicit weight matrices, in Appendix B.3. □

Remark 3.3 (From the asymptotic theorems to standard transformers). Theorems 3.2 and 3.4 use hardmax/ReGLU activations. Softmax/SwiGLU activations approximate hardmax/ReGLU to any required accuracy (Appendix B.6).

3.2 ALMs Are Universal

Despite their small primitive set, ALMs are powerful enough for universal computation. We show that an ALM with only four memories simulates an arbitrary Turing machine, step by step, by emitting one *chain-of-thought token* per simulated machine step: each emitted token records the next instruction, and the running computation lives entirely in the produced token sequence rather than in any hidden state. Composing this with Theorem 3.2 gives us a constructive simulation of any Turing computation by a single autoregressive run of a 2D transformer, at the level of a step-by-step trace rather than at the level of a complexity class.

Theorem 3.4 (ALMs are universal; proof: Appendix B.4)**.** *For every Turing machine M, there is an ALM with $r = 4$ memories that simulates M step by step. Moreover, fixing any universal Turing machine U, this construction yields a single ALM $\mathcal{A}_U$, with constant alphabet and $r = 4$ memories, such that for every Turing machine M and input x, the run of $\mathcal{A}_U$ on the prefix encoding $\langle M, x\rangle$ simulates $U(\langle M, x\rangle)$, and therefore simulates $M(x)$. Combined with Theorem 3.2, the 2D transformer class is Turing complete with a fixed 2D transformer.*

Proof sketch. We reduce through a two-stack pushdown machine, which is itself Turing complete (Minsky, 1967; Sipser, 2012) and is the simpler universal model to target. The simulation is a chain-of-thought: at each position the ALM emits a token recording the next pushdown instruction, and the entire simulated machine state is reconstructed at every step from the prefix of emitted tokens, with no carry-over hidden state. Four memories suffice: we use two cumulative-sum memories to track the running depths D_A, D_B of the two stacks, and two array memories $\mathcal{S}_A, \mathcal{S}_B$ to hold the stack contents indexed by depth.

We make each emitted token carry a constant-arity instruction $(q, \delta_A, \delta_B, u_A, u_B)$ comprising the next control state q, a depth increment $\delta_X \in \{-1, 0, +1\}$ for each stack, and the symbol $u_X \in \Gamma$ pushed onto stack X when $\delta_X = +1$. All five attributes are drawn from finite sets, so the alphabet $|\Sigma_M| = O(|Q|\,|\Gamma|^2)$ is constant.

In one ALM sweep at each position, the emitted token plays *two roles*. First, it carries out the action announced by the previous token: it writes δ_X into the depth memory (so the cumulative sum is the new depth D_X), and when $\delta_X = +1$ it sets $\mathcal{S}_X[D_X] := u_X$, overwriting any prior value at that depth. Second, it decides the next action by looking up the deterministic transition $\delta(q, \mathrm{top}_A, \mathrm{top}_B)$ in a finite table, with the current stack tops read from $\mathcal{S}_A[D_A]$ and $\mathcal{S}_B[D_B]$. The overwrite semantics ensure that the ALM faithfully tracks the simulated stacks: the latest push at each depth wins, and pop and nop positions write at a sentinel index that we never query. Both roles complete in one sweep because the protocol is write-before-read and the memory order is fixed. We give the full details, including how we encode the input on the stacks, in Appendix B.4. □

The construction above establishes universality. We can go further and show that ALMs efficiently simulate word-RAM machines, the standard abstraction of modern computers with unit-cost indexed memory access on w-bit words.

Proposition 3.5 (ALMs simulate word-RAM machines; proof: Appendix B.5)**.** *Fix a w-bit word-RAM model with a fixed instruction set, a constant number of registers, and a main memory of w-bit cells addressed by w-bit indices (so the program, residing in main memory, has at most 2^w instructions). Every such RAM is simulated, step by step, by an ALM with $O(1)$ memories. Two regimes are available depending on how the $O(w)$-bit state (program counter and any deferred register update) is carried across position boundaries:*

- *One ALM position per RAM step at alphabet $|\Sigma| = 2^{O(w)}$, by encoding the carried state as a single token.*

- *$O(w)$ ALM positions per RAM step at alphabet $|\Sigma| = O(w)$, by emitting one symbol per bit and reconstructing the carried state via* CUMSUM *(Appendix B.5).*

Combining Proposition 3.5 with Theorems 3.2 and 2.1 gives the following runtime bound.

Corollary 3.6 (Efficient RAM simulation)**.** *A 2D transformer can simulate T steps of any $O(\log n)$-bit word-RAM machine on an input of size n in $\mathrm{polylog}(n+T)$ amortized time per RAM step.*

4 Turning Programs to Transformer Weights

Section 3 reduces 2D-attention universality to compiling ALMs in principle. In this section, we make it concrete in two contributions: we design *CALM* (Code for Append-only Lookup Machines), a programming language whose programs declare the r memories of an underlying ALM and the constant-depth combinational circuits that drive their keys, values, queries, and the emitted next token; and we build an analytic back-end that compiles a CALM program directly into the weights of a causal 2D transformer. The compiled transformer executes the program in a single autoregressive run.

4.1 The CALM Language and Its Compilation

A CALM program combines two kinds of operations: *memory primitives* that drive a sequence of integer-indexed memories at the current position, and a small *compute vocabulary* that realizes the constant-depth combinational step between them. Each call site of a memory primitive owns a private memory; numbered by program order, these recover the r memories $\mathcal{M}_1, \ldots, \mathcal{M}_r$ of the underlying ALM (Definition 3.1).

Memory primitives, from the ALM. CALM has two atomic memory primitives. $\textsc{LookUp}(q, k, v)$, at the current position i, simultaneously sets $\mathcal{M}[k] := v$ in the call site's private memory $\mathcal{M}$ and returns $\mathcal{M}[q]$, the latest value written there at integer query q, including the current position's write (default zero); $\textsc{CumSum}(v)$ writes v in its own private memory and returns the running sum $\sum_{t \le i} v(t)$. The arguments q, k, v are integer-valued expressions in earlier-computed values. No standalone read or write exists, so atomicity enforces append-only by construction: every read is paired with a write at the same position.

Compute vocabulary. We close the language under two operations: linear combinations $c_1 x + c_2 y + \cdots$ of earlier values with constant rational coefficients, realized as residual-stream wiring; and the gated-ReLU primitive $\textsc{Gate}(a, b) = a\ \mathrm{ReLU}(b)$, realized as one ReGLU neuron. On integer inputs these derive the standard arithmetic vocabulary mechanically (Definition B.4): step indicator, equality, conditional selection, and integer multiplication. These derived operations are convenience macros, not extensions; each reduces to a constant-depth ReGLU+linear network, exactly the form Definition 3.1's compute step requires. A CALM program terminates by declaring a finite collection of token-scoring expressions; the next token is the argmax of those scores, which realizes Definition 3.1's final-circuit token computation.

Gate graph IR. After expansion, a CALM program is a directed acyclic graph $\mathcal{G} = (V_{\text{in}} \cup V_{\text{lookup}} \cup V_{\text{agg}} \cup V_{\text{gate}}, E)$ of three node types (one LOOKUP per memory's read primitive, one AGGREGATE per cumulative-sum primitive, one GATE per ReGLU neuron of the inter-memory compute), wired by linear edges with rational coefficients. This is exactly the gate graph of Definition B.3.

From gate graph to weights. Compiling the gate graph requires deciding which layer and which phase (attention or feed-forward) each gate should occupy, and which residual-stream coordinate should hold its value. Our compiler solves these decisions jointly as a mixed-integer linear program: the MILP encodes the dependencies of the gate graph together with the type-compatibility constraint (LOOKUP and AGGREGATE gates must run in an attention phase, GATE gates in a feed-forward phase) and minimizes the peak number of values that are simultaneously alive. This is exactly the quantity that determines the model's residual width. Where two gates have disjoint lifetimes, the compiler then *reuses* the same residual-stream coordinate, so the model dimension stays small even when the gate graph is large. With the schedule fixed, every weight matrix follows mechanically from the parabolic key encoding for LOOKUP, uniform attention for AGGREGATE, and the ReGLU

coefficients of the compute vocabulary. The output is a standard causal 2D transformer with 2D heads.

Theorem 4.1 (CALM compilation soundness; proof: Appendix C.1)**.** *For every CALM program P the compiled 2D-head transformer reproduces P's runtime next-token sequence exactly on every input (see §3 for the activation-mode discussion).*

4.2 From C to Weights via a CALM-coded WASM Interpreter

CALM (§4.1) is a custom programming language that users can program in directly: any CALM program compiles to weights of a causal 2D transformer by Theorem 4.1. While CALM is sufficient on its own, writing native CALM is unfamiliar to most users, who would prefer to start from existing languages. We therefore build, *as a CALM program*, an interpreter for WebAssembly, a common compilation target for many languages, including C, C++, and Rust.

A WASM interpreter in CALM. We implement a 32-bit integer WebAssembly (Haas et al., 2017) interpreter as a CALM program, covering control flow, the value and call stacks, locals/globals, and byte-addressable linear memory. Our interpreter is restricted to the `i32` subset of WASM operations. All program state lives in the ALM trace as append-only writes indexed by $(\text{depth}, \text{index}, \text{address})$ keys. Compiling this single CALM program with the back-end of §4.1 yields a 2D-attention transformer that interprets any input WASM program: the WASM program and its input are placed in the prefix, and the transformer executes the program step by step in a single autoregressive run.

Folding the WASM program into weights (Futamura). We can go one step further and avoid feeding the WASM program as prefix tokens at all, baking it into the weights instead so that only the input remains in the prompt. We realize the first Futamura projection as a back-end pass that absorbs the static WASM instruction table into a small set of feed-forward neurons whose coefficients encode the program; the rest of the pipeline runs unchanged. Composing these stages gives the end-to-end path $\texttt{C} \xrightarrow{\text{clang}} \text{WASM} \to$ our specializer $\to$ transformer weights.

Implementation, scheduling specifics, the supported WASM subset, and end-to-end verification of the compiled artifact are in Appendix C. We compare with prior weight-construction efforts in §6.

5 Verification

The compiled artifact is a 2D transformer realized in PyTorch with float64 arithmetic, instantiating Definition B.1, with weights produced by the pipeline of §4: a C program is lowered to WebAssembly via `clang`, the WASM interpreter (itself a CALM program) is compiled to weights via the back-end of §4.1, and the result executes the program in one autoregressive run. The formal compiler-correctness guarantee is Theorem 4.1; the tests in this section are regression checks that validate the implementation against independent evaluators, not a substitute for the proof. Our model runs a non-trivial deterministic computation end-to-end (Theorems 3.2 and 3.4) and the asymptotic speedup of the convex-hull KV cache (Appendix A.1) is visible at moderate prefix length.

Correctness. Each compiled program is checked against three reference paths token by token: a reference integer-arithmetic WASM interpreter (the specification), a gate-graph evaluator that computes every gate-graph node's value directly from the prefix without invoking transformer weights (Appendix C.4), and the transformer with a brute-force linear-scan KV cache in place of the convex-hull cache of Appendix A.1. Every test program produces identical traces under all four paths. The corpus exercises every supported instruction: byte-level arithmetic with carry propagation, signed and unsigned comparisons, stack and call manipulation, locals and globals, multi-width memory access, control flow with branches, and recursive function calls. The agreement is the empirical face of CALM compilation soundness (Theorem 4.1).

Capability: an exact Sudoku solver in one autoregressive run. We compile an exact backtracking Sudoku solver, including on published "hardest Sudoku" instances, and run its full execution trace end-to-end inside a single autoregressive PyTorch run. The compiled transformer achieves 100% **exact accuracy** on every puzzle in the test corpus, including the hardest known instances: each emitted token is the program's byte-by-byte execution of the deterministic solver, and the final grid is

correct by construction. This is the regime where state-of-the-art reasoning LLMs cap below 15% on Sudoku-Bench (Seely et al., 2025) and at 0% on the hardest variants, and where Sudoku-specialized architectures (Logical Intelligence, 2026; Giannoulis et al., 2025) reach 97–99% but stop short of perfect. A trace excerpt is shown in Appendix C.3 (Figure 2).

Asymptotic speedup of the hull cache. The convex-hull KV cache (Appendix A.1) reduces lookup attention from linear to logarithmic in the prefix length. The timing experiment exercises the hardmax lookup cache used by the exact compiled executor, not the approximate softmax range-tree structure of Theorem 2.1. At the $\sim$44k-token Collatz trace, the hull cache finishes in 0.99 s while a standard linear-scan KV cache takes 229.2 s on the same trace, a $\sim 230\times$ wall-clock speedup, with the gap widening superlinearly as the prefix grows. Under the same hull cache, the two hardest-Sudoku traces complete in 150.7 s on Inkala's 2012 puzzle (Inkala, 2012) ($\sim$ 5.94M tokens) and 139.4 s on AI Escargot (Inkala, 2006) ($\sim$ 5.36M tokens) on the same machine. Wall-clock numbers are taken on a reference machine running single-threaded PyTorch float64 (CPU-only, no GPU); reproduction instructions are given in Appendix C.4.

6 Related Work

Subquadratic and fine-grained complexity of softmax attention. Tight $N^{2-\Omega(1)}$ SETH lower bounds for the full softmax output exist when entries are above $\sqrt{\log N}$ or when $d = \omega(\log N)$ (Duman Keles et al., 2023; Alman and Song, 2023). Alman and Song (2023) achieve $N^{1+o(1)}$ time only in the bounded-entries regime $\|\cdot\|_\infty = o(\sqrt{\log N})$; Alman and Song (2024) extend the polynomial method to higher orders. Gupta et al. (2026) treat arbitrary entries and any constant d at $\widetilde{O}(N^{2-1/d})$ but do not address the causal setting. The kernel-density-estimation line (Charikar and Siminelakis, 2017; Backurs et al., 2021) estimates partition functions or kernel-vector products. KDEformer (Zandieh et al., 2023) also approximates the full normalized attention output, with spectral-norm guarantees. Approximate methods (Han et al., 2024; Kacham et al., 2024; Katharopoulos et al., 2020; Kitaev et al., 2020; Choromanski et al., 2021) weaken either the model or the guarantee. Theorem 2.1 gives the first online algorithm for 2D attention at arbitrary key/query norms, serving every insertion and every query in amortized polylogarithmic time per token.

Universality of transformer-shaped models. Prior universality results either rely on architectures with a separate input encoder, non-standard scoring, or unbounded precision (Pérez et al., 2021; Bhattamishra et al., 2020), or characterize what causal transformers can in principle compute, often via chain-of-thought (Merrill and Sabharwal, 2024; Nowak et al., 2024; Li and Wang, 2025; Roberts, 2024). Our contribution is complementary: per-step *efficient* simulation and an explicit compilation pipeline. Corollary 3.6 gives a constructive simulation of $O(\log n)$-bit word-RAM machines with $\mathrm{polylog}(n + T)$ amortized time per RAM step, where n is the input size and T the number of simulated steps, and §4 compiles arbitrary integer programs directly into 2D transformer weights; the standard softmax/SwiGLU regime is handled by the finite-horizon bridge of §3. Detailed comparisons are in Appendix D.2.

Constructive programming of transformer weights. Compiling logic into transformer weights has direct precedents: Tracr (Lindner et al., 2023) compiles a RASP (Weiss et al., 2021) subset using bucketed numerical operations, and Looped Transformers (Giannou et al., 2023) compile a universal SUBLEQ into weights through an external looping mechanism; Friedman et al. (2023); Zhou et al. (2024); Yang and Chiang (2024) extract, predict, or compile bounded fragments. Our compiler extends this line: the source is an integer-control-flow language reaching the full `i32` WebAssembly subset (reachable from C through `clang`), the arithmetic is exact rather than bucketed, and the compiled model is a standard causal 2D transformer run autoregressively without external looping, with the program either supplied in the prefix or baked into the weights via the first Futamura projection (§4.2). Detailed comparison in Appendix D.3.

7 Limitations

The precision of the construction scales with the number of simulated steps: T steps of a Turing machine on an input of length n require $O(\log(n + T))$ bits. In our experiments, float64 supports

millions of steps. The softmax/SwiGLU guarantee covers finite runs, with numerical parameters that may depend on the run (Appendix B.6). While our attention algorithm asymptotically speeds up inference for trained models with 2D softmax heads, our experiments focus only on analytically constructed weights.

Acknowledgments and Disclosure of Funding

The authors received no third-party funding or support for this work and declare no competing interests.

References

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. GQA: Training generalized multi-query transformer models from multi-head checkpoints. In *Conference on Empirical Methods in Natural Language Processing*, 2023.

Josh Alman and Zhao Song. Fast attention requires bounded entries. In *Advances in Neural Information Processing Systems*, 2023.

Josh Alman and Zhao Song. How to capture higher-order correlations? Generalizing matrix softmax attention to Kronecker computation. In *International Conference on Learning Representations*, 2024.

Arturs Backurs, Piotr Indyk, Cameron Musco, and Tal Wagner. Faster kernel matrix algebra via density estimation. In *International Conference on Machine Learning*, 2021.

Satwik Bhattamishra, Arkil Patel, and Navin Goyal. On the computational power of transformers and its implications in sequence modeling. In *Conference on Computational Natural Language Learning*, 2020.

Gerth Stølting Brodal and Riko Jacob. Dynamic planar convex hull. In *IEEE Symposium on Foundations of Computer Science*, 2002.

Moses Charikar and Paris Siminelakis. Hashing-based-estimators for kernel density in high dimensions. In *IEEE Symposium on Foundations of Computer Science*, 2017.

Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, et al. Rethinking attention with Performers. In *International Conference on Learning Representations*, 2021.

Feyza Duman Keles, Pruthuvi Mahesakya Wijewardena, and Chinmay Hegde. On the computational complexity of self-attention. In *Algorithmic Learning Theory*, 2023.

Dan Friedman, Alexander Wettig, and Danqi Chen. Learning transformer programs. In *Advances in Neural Information Processing Systems*, 2023.

Angeliki Giannou, Shashank Rajput, Jy-yong Sohn, Kangwook Lee, Jason D Lee, and Dimitris Papailiopoulos. Looped transformers as programmable computers. In *International Conference on Machine Learning*, 2023.

Panagiotis Giannoulis, Yorgos Pantis, and Christos Tzamos. Teaching transformers to solve combinatorial problems through efficient trial & error. In *Advances in Neural Information Processing Systems*, 2025.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. In *Conference on Language Modeling*, 2024.

Shreya Gupta, Boyang Huang, Barna Saha, Yinzhan Xu, and Christopher Ye. Subquadratic algorithms and hardness for attention with any temperature. In *International Conference on Learning Representations*, 2026.

Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017.

Insu Han, Rajesh Jayaram, Amin Karbasi, Vahab Mirrokni, David Woodruff, and Amir Zandieh. HyperAttention: Long-context attention in near-linear time. In *International Conference on Learning Representations*, 2024.

Arto Inkala. AI Escargot, 2006. Sudoku puzzle. http://www.aisudoku.com/index_en.html.

Arto Inkala. The world's hardest Sudoku, 2012. Sudoku puzzle published in The Telegraph, June 2012.

Praneeth Kacham, Vahab Mirrokni, and Peilin Zhong. PolySketchFormer: Fast transformers via sketching polynomial kernels. In *International Conference on Machine Learning*, 2024.

Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are RNNs: Fast autoregressive transformers with linear attention. In *International Conference on Machine Learning*, 2020.

Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. In *International Conference on Learning Representations*, 2020.

Qian Li and Yuyi Wang. Constant bit-size transformers are Turing complete. In *Advances in Neural Information Processing Systems*, 2025.

David Lindner, János Kramár, Sebastian Farquhar, Matthew Rahtz, Tom McGrath, and Vladimir Mikulik. Tracr: Compiled transformers as a laboratory for interpretability. In *Advances in Neural Information Processing Systems*, 2023.

Logical Intelligence. What Sudoku reveals about AI reasoning architectures and the future of our economy, 2026. Blog post, 3 February 2026. https://logicalintelligence.com/blog/energy-based-model-sudoku-demo.

Jiří Matoušek. Efficient partition trees. *Discrete & Computational Geometry*, 8:315–334, 1992.

William Merrill and Ashish Sabharwal. The parallelism tradeoff: Limitations of log-precision transformers. *Transactions of the Association for Computational Linguistics*, 11:531–545, 2023.

William Merrill and Ashish Sabharwal. The expressive power of transformers with chain of thought. In *International Conference on Learning Representations*, 2024.

Marvin L Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, 1967.

Franz Nowak, Anej Svete, Alexandra Butoi, and Ryan Cotterell. On the representational capacity of neural language models with chain-of-thought reasoning. In *Annual Meeting of the Association for Computational Linguistics*, pages 12510–12548, 2024.

Mark H Overmars and Jan van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23(2):166–204, 1981.

Jorge Pérez, Pablo Barceló, and Javier Marinkovic. Attention is Turing-complete. *Journal of Machine Learning Research*, 22(75):1–35, 2021.

Franco P Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

Jesse Roberts. How powerful are decoder-only transformer neural models? In *International Joint Conference on Neural Networks*, 2024.

Jeffrey Seely, Yuki Imajuku, Tianyu Zhao, Edoardo Cetin, and Llion Jones. Sudoku-Bench: Evaluating creative reasoning with Sudoku variants, 2025. arXiv:2505.16135.

Noam Shazeer. Fast transformer decoding: One write-head is all you need, 2019. arXiv:1911.02150.

Hava T Siegelmann and Eduardo D Sontag. On the computational power of neural nets. *Journal of Computer and System Sciences*, 50(1):132–150, 1995.

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

Ryan Spring and Anshumali Shrivastava. A new unbiased and efficient class of LSH-based samplers and estimators for partition function computation in log-linear models, 2017. arXiv:1703.05160.

Gail Weiss, Yoav Goldberg, and Eran Yahav. Thinking like transformers. In *International Conference on Machine Learning*, 2021.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *International Conference on Learning Representations*, 2024.

Andy Yang and David Chiang. Counting like transformers: Compiling temporal counting logic into softmax transformers. In *Conference on Language Modeling*, 2024.

Amir Zandieh, Insu Han, Majid Daliri, and Amin Karbasi. KDEformer: Accelerating transformers via kernel density estimation. In *International Conference on Machine Learning*, 2023.

Hattie Zhou, Arwen Bradley, Etai Littwin, Noam Razin, Omid Saremi, Josh Susskind, Samy Bengio, and Preetum Nakkiran. What algorithms can transformers learn? A study in length generalization. In *International Conference on Learning Representations*, 2024.

A Efficient Algorithms for 2D Attention

This appendix proves Theorem 2.1 and the exact hardmax bound used by the universality construction.

Notation. Let n be the size of a static block, t the current online prefix length, and N an upper bound on the prefix lengths under consideration. We use active width $W_t = \ln(8tV/\varepsilon)$ and Taylor degree $p = O(W_t)$. In softmax complexity bounds below, $\log N$ abbreviates $\log(NV/\varepsilon)$; all bounds are independent of key and query magnitudes.

Computational model. We count real arithmetic, comparisons, and $\exp$/$\log$ evaluations at unit cost. The bounds below concern this operation count, not finite-precision bit complexity. Logarithms in complexity bounds are truncated below at one; the $V = 0$ case is answered in constant time.

A.1 Hardmax via the Convex Hull KV Cache

Given a query $q \in \mathbb{R}^2$ and stored keys $k_0, \ldots, k_i \in \mathbb{R}^2$, hardmax attention returns

$$j^\star \;=\; \arg\max_{j \le i}\, q \cdot k_j.$$

This is the planar *supporting-point query* of computational geometry: given a direction q, return the point in the prefix that is extremal in that direction. A point not on the convex hull of $\{k_0, \ldots, k_i\}$ is a strict convex combination of hull vertices and cannot strictly maximize any linear functional, so only hull keys need to be considered.

When $q_y \neq 0$, the dot product factors as $q \cdot k_j = q_y\big(\frac{q_x}{q_y} k_x^{(j)} + k_y^{(j)}\big)$, so maximizing it reduces to evaluating the upper or lower envelope of the lines $\ell_j(m) = k_x^{(j)} m + k_y^{(j)}$ at slope $m = q_x/q_y$, depending on the sign of q_y. The case $q_y = 0$ reduces to extreme-k_x queries handled by a running min/max. Standard dynamic upper-hull data structures (Overmars and van Leeuwen, 1981; Brodal and Jacob, 2002) maintain the upper envelope online with logarithmic updates and logarithmic queries; we maintain both the upper and the lower envelope (for the two signs of q_y), together with running k_x-extrema for the $q_y = 0$ case. Because the KV cache is append-only, insertion-only maintenance suffices: a key evicted from the hull never returns, so a balanced-tree hull achieves $O(\log N)$ amortized insertion without fully dynamic machinery.

Proposition A.1 (Dynamic 2D hardmax KV cache)**.** *Under online insertion of two-dimensional keys, hardmax attention queries are answered exactly in $O(\log N)$ amortized time per insertion and $O(\log N)$ time per query, where N is the current prefix length, by maintaining the convex hull of the inserted keys.*

Proof. Maintain upper and lower hull chains in balanced search trees ordered by k_x. An insertion locates its neighbors and removes the chain vertices made interior by the new point. Each inserted vertex is removed at most once, so the tree operations total $O(N \log N)$ over N insertions. Binary search on the ordered hull slopes answers a supporting-face query in $O(\log N)$ time.

To average tied values, store sums and counts at duplicate vertices and on the relative interior of each hull edge. A point on an edge updates that edge's aggregate; when an edge is extended, its aggregate is merged with the intervening vertex and collinear-edge aggregates. Aggregates that become strictly interior are discarded. A point that becomes interior never returns to the boundary, so these updates have the same amortized bound. An exposed edge returns its aggregate plus those of its two endpoints; an exposed vertex returns its own aggregate. The collinear case is a segment with the same metadata. For $q = 0$, a global sum and count return the average over all inserted values. Keeping the largest token index instead of a sum gives latest-match tie handling at the same cost. □

Aggregate (uniform) attention. Aggregate heads are the degenerate special case in which all keys are identical, so the hull has size one and insertions and queries are $O(1)$. Aggregate heads are exact under softmax at any scale and require no data-structure support.

A.2 Softmax: Setup and Error Decomposition

Recall that N tokens have keys k_j and queries q_i in $\mathbb{R}^2$ of arbitrary magnitude, with values $v_j \in [-V, V]^d$ for $d = O(1)$. The (vector-valued) softmax output is

$$A_i \;=\; Y_i/Z_i, \qquad Y_i \;=\; \sum_j v_j\, e^{\langle q_i, k_j\rangle}, \qquad Z_i \;=\; \sum_j e^{\langle q_i, k_j\rangle},$$

and the approximation condition matches Theorem 2.1: $\|\hat{A}_i - A_i\|_\infty \le \varepsilon$. We target an additive ℓ_∞ guarantee per coordinate; relative error in Y_i is not the right invariant because v_j can be signed and $Y_i^{(c)}$ may pass through zero or change sign through cancellation.

We achieve the additive output bound via a *denominator-relative / numerator-additive* pair. An internal truncation budget $\eta > 0$ controls

$$|\hat{Z}_i - Z_i| \;\le\; \eta\, Z_i, \tag{1}$$

$$\left|\hat{Y}_i^{(c)} - Y_i^{(c)}\right| \;\le\; \eta\, V\, Z_i \quad \text{for each coordinate } c. \tag{2}$$

Set the *active width*

$$W \;=\; \ln(8NV/\varepsilon),$$

and the internal budget $\eta = \varepsilon/(4V)$. Throughout we assume the *nontrivial regime* $0 < \varepsilon < 2V$ (in particular $V > 0$), which makes $W \ge \ln 4$ immediate for $N \ge 1$: otherwise every attention output lies in $[-V, V]^d$ coordinatewise, so the query algorithm returns 0 without consulting any structure — already an ε-approximation (this also covers $V = 0$). Online queries use $N = t$, hence $W_t = \ln(8tV/\varepsilon)$. Section A.4 shows how to choose each block's stored degree to cover every query during its lifetime.

Normalization. Since $|Y_i^{(c)}| \le V Z_i$ and $\hat{Z}_i \ge (1-\eta)Z_i$, (1)–(2) imply

$$\left|\frac{\hat{Y}_i^{(c)}}{\hat{Z}_i} - \frac{Y_i^{(c)}}{Z_i}\right| \le \frac{2\eta V}{1-\eta} \le 4\eta V = \varepsilon, \tag{3}$$

where $\eta = \varepsilon/(4V) < 1/2$. It remains to obtain these two estimates.

Lemma A.2 (Truncation error)**.** *Suppose the retained set S contains every key with score at least $m_i - W$, and every retained key has score at least $m_i - cW$, where $m_i = \max_j \langle q_i, k_j\rangle$ and $c \ge 1$ is constant. Replacing $e^{\langle q_i, k_j\rangle - m_i}$ on S by its degree $p = \lceil 4(c+1)W\rceil$ Taylor polynomial gives estimates satisfying (1)–(2) with $\eta = \varepsilon/(4V)$.*

Proof. The omitted exponential mass is at most $Ne^{m_i - W} \le (\varepsilon/(8V))Z_i$. Its value-weighted mass is at most V times this bound in each coordinate.

For retained keys the shifted score lies in $[-cW, 0]$. Put $R = cW$. Stirling's inequality and $p+1 > 4R$ bound the absolute Taylor remainder by

$$\sum_{a>p} \frac{R^a}{a!} \le \frac{4}{3}\left(\frac{eR}{p+1}\right)^{p+1} \le \frac{4}{3}e^{-1.5W} \le e^{-W} = \frac{\varepsilon}{8NV}.$$

Here $p + 1 > 4W$ and $4(\ln 4 - 1) > 1.5$ give the middle inequality, and $W \ge \ln 4$ gives the last. Summing over at most N retained keys and multiplying by $e^{m_i} \le Z_i$ gives another $(\varepsilon/(8V))Z_i$ denominator error and $(\varepsilon/8)Z_i$ error per numerator coordinate. Adding the two contributions proves the bounds; (3) gives the output guarantee. □

A.3 Static Query Against a Fixed Key Set

We construct, for an arbitrary fixed *indexed* key collection $K = (k_j)_{j\in B} \subset \mathbb{R}^2$ of size n — a multiset: distinct token indices may carry identical coordinates, sums over $k \in K$ range over token indices, and the range tree stores indexed records — a static data structure that answers any query $q \in \mathbb{R}^2$ for $Z(q) = \sum_{k\in K} e^{\langle q,k\rangle}$ and the value-weighted analog $Y(q)$ in $O(\log^4 N)$ time so that the resulting $\hat{A}(q) = \hat{Y}(q)/\hat{Z}(q)$ satisfies $\|\hat{A}(q) - A(q)\|_\infty \le \varepsilon$. The construction is built in $O(n \log^3 N)$ time and uses $O(n \log^3 N)$ space.

A.3.1 Sector Decomposition and Hull Shifting

Partition queries by dominant coordinate into four sectors, $S_1 = \{q : q_y \geq |q_x|\}$ and three symmetric variants. We treat S_1 explicitly; the other three are obtained by the symmetric reflections of the construction below, and we maintain one data structure per sector. (The degenerate query $q = (0, 0)$ gives every key the same score, so its attention output is the plain prefix average, returned directly without consulting the structure.)

Within S_1 write $q = q_y(\lambda, 1)$ with $\lambda = q_x/q_y \in [-1, 1]$ and $q_y > 0$, so $\langle q, k\rangle = q_y(\lambda k_x + k_y)$. Let $h_x : [k_x^{\min}, k_x^{\max}] \to \mathbb{R}$ denote the upper hull of K, i.e. the concave piecewise-linear function whose graph is the upper boundary of $\mathrm{conv}(K)$; every $k \in K$ satisfies $k_y \leq h_x(k_x)$. Define the support function in the slope variable $h^*(\lambda) = \max_{k\in K}(\lambda k_x + k_y) = \max_x(\lambda x + h_x(x))$, attained at a hull vertex $x^\star(\lambda)$ whose adjacent slopes bracket $-\lambda$, or at an endpoint. If a whole edge maximizes the score, choose either endpoint. Then $m_q = q_y \cdot h^*(\lambda)$ and the optimum is the hull vertex $k^\star(\lambda) = (x^\star(\lambda), h_x(x^\star(\lambda)))$, located in $O(\log n)$ by binary search on hull edges sorted by slope.

The shift. Translate every key by the upper-hull function in the y-direction:

$$(k_x, k_y) \;\longmapsto\; (k_x,\, y'), \qquad y' := k_y - h_x(k_x) \leq 0.$$

After the shift, every key has $y' \leq 0$, with equality only on the upper hull.

A.3.2 Sandwich Lemma

The slack of the Taylor surrogate (Lemma A.2) lets us replace the exact active set by an axis-aligned rectangle in shifted coordinates.

Lemma A.3 (Active-region rectangle)**.** *Let $g(x) := \lambda x + h_x(x)$, a concave function of $x \in [k_x^{\min}, k_x^{\max}]$ attaining its maximum $h^*(\lambda)$ at $x^\star(\lambda)$, and let $[x^{\mathrm{lo}}, x^{\mathrm{hi}}]$ be the unique interval of x on which $g(x) \geq h^*(\lambda) - W/q_y$. Define the rectangle in shifted coordinates*

$$R(q) \;:=\; [x^{\mathrm{lo}}, x^{\mathrm{hi}}] \times [-W/q_y, 0].$$

Then

$$\{k \in K : \langle q, k\rangle \geq m_q - W\} \;\subseteq\; \{k \in K : k \in R(q)\} \;\subseteq\; \{k \in K : \langle q, k\rangle \geq m_q - 2W\}.$$

Proof. Active set is contained in the rectangle. If $\langle q, k\rangle \geq m_q - W$, then $\lambda k_x + k_y \geq h^*(\lambda) - W/q_y$, equivalently $y' + g(k_x) \geq h^*(\lambda) - W/q_y$. Since $g(k_x) \leq h^*(\lambda)$ and $y' \leq 0$, both inequalities $y' \geq -W/q_y$ and $g(k_x) \geq h^*(\lambda) - W/q_y$ follow, the latter giving $k_x \in [x^{\mathrm{lo}}, x^{\mathrm{hi}}]$.

Rectangle is contained in the relaxed active set. If $k \in R(q)$, then $y' \geq -W/q_y$ and $g(k_x) \geq h^*(\lambda) - W/q_y$, so $\lambda k_x + k_y = y' + g(k_x) \geq h^*(\lambda) - 2W/q_y$, hence $\langle q, k\rangle \geq m_q - 2W$. □

By Lemma A.2, replacing the exact active set by $R(q) \cap K$ in the Taylor approximation preserves the additive output error bound ε at the cost of taking the Taylor degree to $p = \lceil 4 \cdot (2W + W)\rceil = O(W)$, the same order as for the exact active set.

A.3.3 2D Rectangle Range Tree with Augmented Moments

By Lemma A.3, the kernel sum reduces to a sum over $R(q) \cap K$, an axis-aligned rectangular range in shifted coordinates. We answer it with a standard 2D range tree (Preparata and Shamos, 1985). Crucially, the range tree is *indexed* by shifted coordinates (k_x, y') so that the rectangle query $R(q)$ becomes axis-aligned, while the *moments stored* at each canonical node are over the *original* key coordinates (k_x, k_y).

Range tree with augmented moments. A 2D range tree on the n shifted keys is a primary 1D segment tree on the k_x-coordinate; at each canonical node u of the primary tree, a secondary 1D segment tree is built on the y'-coordinates of the keys K_u owned by u. At every canonical node u' of the secondary tree, with $K_{u'} \subseteq K_u$, precompute the bivariate origin-centered moments of the original key coordinates:

$$M_{a,b}^{(u')} \;:=\; \sum_{k\in K_{u'}} k_x^a\, k_y^b, \qquad 0 \leq a + b \leq p, \tag{4}$$

together with the value-weighted moments $M_{a,b}^{(u'),v_c} = \sum_{k \in K_{u'}} v^{(c)}(k)\, k_x^a\, k_y^b$ for each value coordinate c. Each canonical node stores $O(W^2)$ moments per value channel ($d+1$ channels in total: one for Z and d for Y); since $d = O(1)$ this is $O(W^2)$ moments per node.

Construction cost. A 2D range tree on n keys has $O(n \log n)$ canonical nodes total across the secondary trees. At each, the $O(W^2)$ moments are computed in $O(W^2)$ time by summing the two children's moments (the bivariate moments share the common origin and require no polynomial shift in construction). Total moment construction is $O(n \log n \cdot W^2) = O(n \log^3 N)$, dominating the $O(n \log n)$ upper-hull build.

Answering a query. Given $q \in S_1$, locate $\lambda = q_x/q_y$ on the hull ($O(\log n)$), compute $h^*(\lambda)$ and the supporting key $k^\star = (k_x^\star, k_y^\star)$, and read off $x^{\mathrm{lo}}, x^{\mathrm{hi}}$ from the hull breakpoints at which g falls below $h^*(\lambda) - W/q_y$, interpolating on the crossing edges when needed ($O(\log n)$). Issue a 2D range query on the range tree for the rectangle $R(q) = [x^{\mathrm{lo}}, x^{\mathrm{hi}}] \times [-W/q_y, 0]$ in shifted coordinates; this returns $R(q) \cap K$ as a disjoint union of $O(\log^2 n)$ canonical subsets $K_{u'}$, regardless of the rectangle's dimensions.

For $E_r(z) = \sum_{j=0}^{r} z^j/j!$, define

$$C_{a,b}(q) = \frac{q_x^a q_y^b}{a!b!}\, E_{p-a-b}(-m_q), \qquad 0 \le a+b \le p.$$

The multinomial theorem gives the exact polynomial identity

$$\sum_{s=0}^{p} \frac{(q_x k_x + q_y k_y - m_q)^s}{s!} = \sum_{a+b \le p} C_{a,b}(q) k_x^a k_y^b.$$

Thus the contribution of a canonical subset is

$$\hat{Z}_{u'}(q) = e^{m_q} \sum_{a+b \le p} C_{a,b}(q) M_{a,b}^{(u')}. \tag{5}$$

The same coefficients contracted with $M^{(u'),v_c}$ give $\hat{Y}_{u'}^{(c)}$. Compute all $E_r(-m_q)$ by successive summation and all powers and factorials by recurrence. This takes $O(p)$ operations, followed by $O(p^2)$ operations to form the coefficient table once per query. Each canonical subset then costs $O(p^2)$ per value coordinate. For normalized estimates, omit the common factor e^{m_q}; it cancels in the output ratio.

Lemma A.4 (Static query cost)**.** *The structure on n keys has total construction cost $O(n \log^3 N)$ and answers every query so that $\|\hat{A}(q) - A(q)\|_\infty \le \varepsilon$ in $O(\log^4 N)$ time, independent of the magnitudes of the keys and queries.*

Proof. Construction was bounded above. For the query cost: hull lookup and rectangle determination cost $O(\log n)$; forming the Taylor coefficients costs $O(W^2)$ once per query; the 2D rectangular range query returns $O(\log^2 n)$ canonical subsets, each contributing an $O(W^2)$ contraction of its origin-centered moments (4) against these coefficients; the total is $O(W^2(1 + \log^2 n)) = O(\log^4 N)$. Correctness: Lemma A.3 places $R(q) \cap K$ between the active set and the $2W$-relaxed active set, so Lemma A.2 guarantees that the resulting $(\hat{Z}, \hat{Y})$ satisfy (1)–(2) with $\eta = \varepsilon/(4V)$; Equation (5) computes the truncated Taylor expansion of $e^{\langle q,k \rangle}$ in the original coordinates over $R(q) \cap K$ exactly by the multinomial identity above. The output bound follows from Equation (3). □

A.4 Online via the Doubling Trick

Inserting a key can change the hull shift of existing keys and hence their range-tree coordinates. We avoid updating the static structure of §A.3 by a doubling trick that maintains $O(\log N)$ static structures in parallel and rebuilds them by a binary-counter pattern.

Layout. At any prefix length t, let $t = \sum_{\ell \in B(t)} 2^\ell$ be the binary expansion of t. Maintain one static structure $\mathcal{D}_\ell$ for every $\ell \in B(t)$; $\mathcal{D}_\ell$ covers a contiguous block of 2^ℓ keys. Together, $\{\mathcal{D}_\ell\}_{\ell \in B(t)}$ partition the prefix of inserted keys.

Insertion. On insertion of the $(t+1)$-th key, the binary expansion advances from t to $t+1$. Standard binary-counter mechanics: a (possibly empty) suffix of low-order bits flips from 1 to 0, and one higher-order bit flips from 0 to 1. Concretely, every flipped $\mathcal{D}_\ell$ is discarded and merged into a single new $\mathcal{D}_{\ell^*}$ of size 2^{ℓ^*} that is built from scratch on the union of their keys plus the new key. No prior knowledge of the horizon N is required: the merged block at time $t+1$ is built using the current width $W_{t+1} = \ln(8(t+1)V/\varepsilon)$ and the explicit over-provisioned Taylor degree $p_{t+1} = \lceil 12\,(W_{t+1} + \ln 2)\rceil$ (Lemma A.2 at $c = 2$ requires degree $\lceil 12\,W_t\rceil$ at any time t the block serves a query, and $W_t < W_{t+1} + \ln 2$ throughout its lifetime, as shown next). A level-ℓ^* block built at time t' serves queries only while bit ℓ^* remains set, i.e. for $t < t' + 2^{\ell^*} \le 2t'$, so the query-time width exceeds the build-time width by less than $\ln 2$ and the $\Theta(\cdot)$ slack in the degree covers every query the block ever answers.

At level ℓ, there are $O(N/2^\ell)$ rebuilds, each costing $O(2^\ell \log^3 N)$ by Lemma A.4. Summing over $O(\log N)$ levels gives $O(N \log^4 N)$ total construction work, or $O(\log^4 N)$ amortized per insertion.

Query. A query at time t is answered by running the static query of Lemma A.4 on each $\mathcal{D}_\ell$ for $\ell \in B(t)$ and combining the contributions to Z and Y. Because each static block uses its own local maximum $m_q^{(\ell)} = \max_{k \in K_\ell} \langle q, k\rangle$ to factor out the exponential prefactor, we combine blocks using the standard log-sum-exp shift to a common maximum, so that all exponentiated score differences are non-positive: each block returns the triple $(m_q^{(\ell)},\ \hat{Z}_\ell,\ \hat{Y}_\ell)$ with $\hat{Z}_\ell \approx e^{-m_q^{(\ell)}} Z_{i,\ell}$ and $\hat{Y}_\ell \approx e^{-m_q^{(\ell)}} Y_{i,\ell}$; then

$$M \;:=\; \max_{\ell \in B(t)} m_q^{(\ell)}, \qquad \hat{Z}_i \;=\; \sum_\ell e^{m_q^{(\ell)} - M} \hat{Z}_\ell, \qquad \hat{Y}_i \;=\; \sum_\ell e^{m_q^{(\ell)} - M} \hat{Y}_\ell,$$

and the output is $\hat{A}_i = \hat{Y}_i / \hat{Z}_i$. The shift makes every exponentiated argument non-positive and the algebraic identity $Z_i = \sum_\ell Z_{i,\ell}$, $Y_i = \sum_\ell Y_{i,\ell}$ is preserved exactly in the real-arithmetic model.

Proof of Theorem 2.1. Insertion costs $O(\log^4(NV/\varepsilon))$ amortized by the preceding rebuild count. For queries, each $\mathcal{D}_\ell$ contributes $O(\log^4 N)$ by Lemma A.4, and there are $|B(t)| = O(\log N)$ live structures, giving $O(\log^5(NV/\varepsilon))$ time per query. For correctness, the union of the live structures' key sets equals the prefix, and each block's estimates satisfy the denominator-relative/numerator-additive pair (1)–(2) against its own partial sums (Lemma A.2); the pair is preserved exactly by the log-sum-exp rescaling above (each block's triple is scaled by the positive factor $e^{m_q^{(\ell)} - M}$) and is closed under summation, so the combined $(\hat{Z}_i, \hat{Y}_i)$ satisfy (1)–(2) against the global Z_i, Y_i, and Equation (3) bounds the combined ratio by ε. (Linearity is applied to numerator and denominator errors, never to per-block attention ratios.) □

B Simplified Transformer, ALM, and Universality

§B.1 fixes the simplified causal 2D transformer used throughout the proofs (a 2D transformer pared down to the components the construction needs), and §B.6 relates it to the softmax/SwiGLU model through a finite-horizon bridge. §B.2 introduces the Append-only Lookup Machine (ALM) as an abstraction of per-token transformer computation: a gate graph with three primitive types (AGGREGATE, LOOKUP, and GATE) that pins down a sufficient primitive set for universal computation. §B.3 proves the constructive realization in transformer weights (Theorem 3.2), and §B.4 proves universality of the ALM class (Theorem 3.4).

B.1 The Simplified Transformer Model

We work with a simplified causal transformer containing the components used by the proofs. Appendix B.6 justifies replacing hardmax/ReGLU by sufficiently sharp softmax/SwiGLU on a fixed finite horizon. The model omits LayerNorm and RMSNorm and supplies the positional features below explicitly; we do not prove a reduction from architectures with normalization or RoPE to this model.

Definition B.1 (Simplified causal transformer)**.** Fix a vocabulary Σ, depth L, residual width d_{model}, and feed-forward width d_{ffn}. A simplified causal transformer consists of:

- an embedding matrix $E \in \mathbb{R}^{|\Sigma| \times d_{\text{model}}}$;
- L causal multi-head attention layers with head dimension $d_h = 2$;
- L gated feed-forward blocks of width d_{ffn} of the form $x \mapsto x + W_{\text{down}}(\text{ReLU}(W_{\text{gate}}x) \odot W_{\text{up}}x)$; and
- an output matrix $W_{\text{out}} \in \mathbb{R}^{|\Sigma| \times d_{\text{model}}}$.

Built-in scalar features. In addition to the (position-independent) token embedding, the simplified model includes an explicit *positional-feature map* that deterministically writes four built-in scalars into reserved residual columns at position i (this map is part of Definition B.1; we do not claim these exact features are derivable from standard mechanisms such as RoPE):

$$\mathbf{1}(i) = 1, \qquad \mathbf{pos}(i) = i, \qquad \lambda(i) = \frac{1}{\log 2} - \frac{1}{\log(i+2)}, \qquad \mathbf{pos}^2(i) = i^2.$$

The function λ is monotone increasing and uniformly bounded by $1/\log 2$. It is used only to break ties in favor of later positions without overwhelming the unit score gap created by the parabolic key encoding. Any uniformly bounded, strictly increasing sequence works in its place. For the finite-horizon *rational-arithmetic* realization (Appendix B.3, Numerical scope) we instantiate the tie-break with the rational sequence $\lambda(i) = i/(i+1)$ — strictly increasing, in $[0, 1)$, exactly representable in $O(\log N)$ bits over a prefix of N tokens, with adjacent gaps $\Theta(1/i^2)$, still $O(\log N)$ bits of margin. The logarithmic form remains an option for the exact real-arithmetic model; the bit bound uses the rational form.

Attention semantics. The exact theorems use Definition B.1 with two head types:

- **LOOKUP heads** use causal hard attention, averaging the values at all maximizing positions $j \le i$ (the current position included). In the lookup construction, the positional perturbation makes the latest visible match the unique maximizer.
- **AGGREGATE heads** are uniform-average heads: all logits are equal, so the output is the average of the value vectors over the prefix. This behavior is exact under ordinary softmax.

The compiled construction matches this hybrid description: the softmax artifact approximates LOOKUP heads by very sharp attention, while AGGREGATE heads are exact at every scale. Head outputs are concatenated and passed through the linear output projection, which routes them to the designated residual coordinates.

Implementation. The public implementation (Appendix C.4) uses float64, sharp softmax lookup heads, and ReGLU gates. The optional SwiGLU replacement is covered by Appendix B.6. Appendix C.2 describes scheduling and residual-slot reuse; these optimizations preserve the construction above.

B.2 The Append-only Lookup Machine

The proofs use the following *gate-graph* presentation of Definition 3.1. Appendix B.3 implements its three primitives in transformer weights.

By universality, an ALM can also simulate any finite-precision 2D transformer. This does not give an efficient translation of transformer layers into ALM primitives; the compilation proved here goes from ALMs to transformers.

Definition B.2 (Expression)**.** An *expression* is a finite $\mathbb{Q}$-linear combination of gate-graph nodes. If $e = \sum_d c_d d$, then its value at position i is $e(i) = \sum_d c_d d(i)$.

Definition B.3 (ALM gate graph)**.** The gate graph is organized into layers $0, 1, \ldots, L$. Layer 0 is the input: token-specific features and the two positional features $\mathbf{1}(i) = 1$ and $\mathbf{pos}(i) = i$. Every non-input node is one of the following types.

(i) **AGGREGATE.** An AGGREGATE returns the prefix average

$$A(i) = \frac{1}{i+1} \sum_{j=0}^{i} f(j),$$

where f is an expression in earlier-layer nodes.

(ii) **LOOKUP.** A LOOKUP is specified by an integer-valued query expression q, an integer-valued key expression k, value expressions $v_1, \ldots, v_m$, and an optional visibility mask $\mu \in \{0, 1\}$. It returns the values most recently written under key $q(i)$ at a visible position $j \leq i$ — the current position's own write included, matching the write-before-read protocol of Definition 3.1 — and zero when no such position exists. μ defaults to identically 1.

(iii) **GATE.** A GATE returns

$$G(i) = a(i)\ \mathrm{ReLU}(b(i)),$$

where a and b are expressions in earlier-layer nodes (and in current-layer AGGREGATE or LOOKUP outputs from the immediately preceding attention sub-phase, never in other current-layer GATE outputs).

Lookup arguments are expressions in earlier-layer nodes. All node types and counts are fixed by the program and independent of position.

The ALM emits the next token by argmax over a finite family of output scoring expressions in the layer-L nodes; ties are broken deterministically toward the lowest token index. Exact compilation uses the same rule, whether or not the maximizer is unique.

Memory-sweep equivalence. For each memory in Definition 3.1, all lookup reads use the same write expressions k, v, and each read supplies its own query. Their private copies therefore contain identical values. A cumulative-sum read uses an AGGREGATE on the same written values followed by multiplication by $\mathbf{pos}+\mathbf{1}$. The intervening circuits become GATE nodes. Thus r counts memories in the source machine, not its individual read nodes. To suppress a write, encode visible keys and queries as $2k, 2q$ and use the never-queried key 1 for a masked write. This realizes the optional mask without assuming an unused original address.

Definition B.4 (Derived operations)**.** For integer-valued expressions:

$$\begin{aligned} \mathrm{step}(c) &= \mathrm{ReLU}(c + \mathbf{1}) - \mathrm{ReLU}(c), \\ \mathrm{gated}(g, c) &= g\ \mathrm{step}(c), \\ \mathrm{eq}(e, v) &= \mathrm{step}(e - v) - \mathrm{step}(e - v - \mathbf{1}). \end{aligned}$$

For any integer b,

$$\mathrm{multiply}(a, b)\ =\ a\ \mathrm{ReLU}(b) - a\ \mathrm{ReLU}(-b)\ =\ ab$$

(a single GATE suffices when $b \geq 0$ is guaranteed).

On integers, $\mathrm{step}(c) = \mathbf{1}[c \geq 0]$ and $\mathrm{eq}(e, v) = \mathbf{1}[e = v]$. Conditional selection is $u\mathbf{1}[C] + v(1 - \mathbf{1}[C])$. Each formula uses a constant number of linear maps and GATE nodes.

B.3 Proof of Theorem 3.2: 2D Transformers Implement ALMs

We implement the gate graph of Appendix B.2 in topological order, storing each computed value in a fresh residual column. Uniform attention implements AGGREGATE, and a ReGLU neuron implements GATE. The only nontrivial primitive is LOOKUP.

Parabolic lookup. For integer keys and queries,

$$2qk - k^2 = q^2 - (k - q)^2.$$

Thus an exact match beats every non-match by at least one before tie-breaking. Choose $0 < \alpha < \log 2$ and use the bounded increasing feature λ of Appendix B.1, so that $0 \leq \alpha\lambda(j) < 1$. A visibility penalty excludes masked positions whenever a visible match exists.

Lemma B.5 (Latest visible exact match wins under the parabolic encoding)**.** *Assume $0 < \alpha < \log 2$ and $B > \alpha \sup_j \lambda(j)$. If some visible position has key $q(i)$, the maximizer over $j \leq i$ of*

$$2q(i)k(j) - k(j)^2 + \alpha\lambda(j) - B(1 - \mu(j))$$

is the latest visible position with key $q(i)$.

Proof. A visible match scores at least $q(i)^2$. Every visible non-match scores at most $q(i)^2 - 1 + \alpha \sup \lambda < q(i)^2$, and every masked position scores at most $q(i)^2 + \alpha \sup \lambda - B < q(i)^2$. Among visible matches, strict monotonicity of λ selects the latest. □

We use $B = 1$ in the compilation. This also makes the maximizer unique when no visible match exists: each score is an integer plus $\alpha\lambda(j) \in [0, 1)$. Unequal integer parts cannot tie, and equal integer parts are distinguished by the strictly increasing positional term. Consequently hardmax averaging always returns one position in these lookup heads.

Lemma B.6 (LOOKUP implementation)**.** *Every scalar ALM* LOOKUP *is implemented exactly by one 2D hardmax attention head and a constant-size ReGLU circuit. Fixed-width vector values are implemented coordinatewise.*

Proof. First materialize $k^2 = k\,\mathrm{ReLU}(k) + (-k)\,\mathrm{ReLU}(-k)$ and $\mu v = v\,\mathrm{ReLU}(\mu)$ using ReGLU neurons. The linear projections

$$\begin{aligned} Q(i) &= (q(i), 1), \\ K(j) &= (2k(j), -k(j)^2 + \alpha\lambda(j) - (1 - \mu(j))), \\ V(j) &= (\mu(j)v(j), k(j)) \end{aligned}$$

then return $(\mu(j^\star)v(j^\star), k(j^\star))$ at the unique maximizing position $j^\star$. Apply the integer equality circuit of Definition B.4 to obtain

$$\mathrm{out}(i) = \mathrm{eq}(k(j^\star), q(i))\,\mu(j^\star)v(j^\star). \tag{6}$$

If a visible match exists, Lemma B.5 gives its latest value. Otherwise every candidate is masked or has the wrong key, so one of the two factors preceding $v(j^\star)$ is zero. The equality circuit and final product take a constant number of FFN phases; all their inputs are available before they are used. □

Proof of Theorem 3.2. The embedding supplies the token attributes and positional features. Maintain the invariant that each completed gate-graph node has its exact value in a dedicated residual column at every prefix position. For an AGGREGATE with value expression f, choose $Q = K = 0$ and $V = f$: its output is exactly $(i + 1)^{-1} \sum_{j \le i} f(j)$. Multiplication by $i + 1$ gives a cumulative sum when required. Lemma B.6 implements each LOOKUP, and one ReGLU neuron computes $a\,\mathrm{ReLU}(b)$ for each GATE.

Process nodes in topological order, inserting the constant number of phases needed by each lookup. Each operation reads only already materialized values, and its output projection writes to a fresh, initially zero column. Residual addition therefore preserves every earlier value and establishes the invariant for the next node. The graph is fixed, so the resulting depth and width are constant in the prefix length.

Finally materialize the finitely many output scoring expressions and let W_{out} select these columns. The same deterministic argmax rule therefore emits the same next token. For quadratic scores, the identity

$$-(p_k - \hat{v}_k)^2 = -p_k^2 + 2\hat{v}_k p_k - \hat{v}_k^2$$

allows all token rows to share the columns p_k, p_k^2 instead of materializing one column per token. Induction over generated positions, starting from any supplied prefix, proves equality of the complete autoregressive sequences. □

Numerical scope. Use the rational tie-break $\lambda(i) = i/(i + 1)$ with $\alpha = 1/2$. Over N positions its adjacent score gaps are at least $1/(2N(N + 1))$. If M_N bounds the absolute values of the ALM's integer keys, values, and circuit intermediates, the construction uses $O(\log(2 + M_N) + \log(N + 1))$ bits per rational coordinate. Indeed, squaring a key only doubles its bit length; a prefix sum of integer values is bounded by NM_N; its temporary average has denominator at most N and is multiplied by the prefix length before further ALM computation. The remaining fixed number of arithmetic operations and fixed rational coefficients change bit lengths by only a construction-dependent constant factor.

For the two-stack simulation, $N = O(n + T)$, keys are depths at most $n + T$, and symbols come from a finite alphabet. Its fixed circuit therefore requires $O(\log(n + T))$ bits, or $O(\log T)$ when

$n = O(T)$. The w-bit word-RAM construction similarly uses $O(w + \log(N+1))$ bits. More generally, a fixed ALM has polynomially bounded integer intermediates: input attributes are bounded, position is at most N, lookup selects an earlier-stage value, cumulative sum adds at most a factor N, and multiplication adds polynomial degrees. Induction over the fixed circuit and memory order proves the bound. This is an exact rational representation bound; it does not assert exact execution in floating-point arithmetic.

B.4 Proof of Theorem 3.4: ALMs Are Turing Complete

We simulate a two-stack pushdown machine with four memories: two cumulative sums for stack depths and two lookups for stack contents. Each simulated transition emits one token.

Definition B.7 (Two-stack pushdown machine)**.** A *two-stack pushdown machine* is a tuple $M = (Q, \Gamma, \delta, q_0, F)$ with finite control-state set Q, finite stack alphabet Γ containing a distinguished bottom marker $\bot$, initial state $q_0 \in Q$, accepting set $F \subseteq Q$, and deterministic transition function

$$\delta : Q \times \Gamma \times \Gamma \longrightarrow Q \times \mathrm{Act} \times \mathrm{Act},$$

where the action set is $\mathrm{Act} = \{\texttt{push}_u : u \in \Gamma \setminus \{\bot\}\} \cup \{\texttt{pop}, \texttt{nop}\}$ (the bottom marker is never pushed as data). The top symbol of an empty stack is $\bot$. Without loss of generality, δ never schedules a pop when the corresponding stack top is $\bot$. Two-stack pushdown machines are Turing complete (Minsky, 1967; Sipser, 2012). A direct simulation stores the current tape symbol in the finite control, and the symbols to its left and right in the two stacks, nearest symbol on top. A head move pushes the written symbol onto one stack and pops the other, using a blank when that stack is empty. Thus one Turing-machine transition needs one two-stack transition.

Proof of Theorem 3.4. Fix any two-stack machine $M = (Q, \Gamma, \delta, q_0, F)$. We construct an ALM $\mathcal{A}_M$ whose emitted token stream is, position by position, the chain-of-thought trace of M on its input. We use the gate-graph formalization of Appendix B.2; the construction uses $r = 4$ memories (two AGGREGATEs for stack depths and two LOOKUPs for stack contents) plus a constant number of GATE nodes for the transition logic.

Token alphabet. Each token w_i carries five finite attributes:

$$w_i = \big(q(i),\ \delta_A(i),\ \delta_B(i),\ u_A(i),\ u_B(i)\big),$$

where

- $q(i) \in Q$ is the control state of the simulated machine once the action carried by token i has been committed (equivalently: the state entered by the transition that emitted token i);
- for each stack $X \in \{A, B\}$, the depth increment $\delta_X(i) \in \{-1, 0, +1\}$ (with -1 for pop, $+1$ for push, 0 for nop);
- $u_X(i) \in \Gamma$ is the symbol pushed onto stack X at position i, used only when $\delta_X(i) = +1$ (a fixed dummy otherwise).

The alphabet is bounded by $|\Sigma_M| \le |Q| \cdot 9 \cdot |\Gamma|^2$, hence finite. The start token w_0 encodes $q(0) = q_0$, $\delta_X(0) = 0$, $u_X(0) = \bot$, providing the bottom-marker sentinel for both stacks.

Stack-depth memory. For each stack X, an AGGREGATE followed by a multiply-by-$\mathbf{pos} + \mathbf{1}$ GATE computes the running depth

$$D_X(i) \ = \ \sum_{t=0}^{i} \delta_X(t) \ \ge \ 0,$$

the number of symbols currently on stack X. Non-negativity is preserved because δ never schedules a pop on $\bot$ (Definition B.7).

Stack-contents memory. A LOOKUP stores the contents of each stack indexed by its push-time depth. For stack X, define

$$\kappa_X(i) = D_X(i), \qquad \mu_X(i) = \mathbf{1}[\delta_X(i) = +1] \quad (\text{for } i > 0),$$

with the start-token sentinel $\kappa_X(0) = 0$, $\mu_X(0) = 1$, $u_X(0) = \bot$. The mask μ_X makes only push events visible to the lookup; pop and nop positions are masked out.

Stack-top retrieval. After committing the current token's writes, query each contents memory at $D_X(i)$.

Lemma B.8 (Stack invariant)**.** *At every position i, the value retrieved by the stack-contents lookup with query $D_X(i)$ equals the top symbol of stack X in the simulated computation of M.*

Proof. Induct on the stronger invariant that, for every depth $1 \le d \le D_X(i)$, the latest visible write at key d stores the symbol at depth d of the current stack, and key 0 stores $\perp$. The start token establishes the base case. A push writes the new top at depth $D_X(i-1)+1$ and leaves all lower entries unchanged. A no-op changes neither the stack nor any visible write. A pop makes no visible write and restricts the invariant to the remaining depths. Lemma B.5 therefore returns the top symbol at the new depth, including $\perp$ at depth zero. □

Transition logic. The transition function δ has finite domain $Q \times \Gamma^2$, so it is fully described by a finite table of $|Q|\,|\Gamma|^2$ entries. Each entry is selected by a product of equality indicators

$$\chi_{q,a,b}(i) \;=\; \mathbf{1}[q(i)=q]\,\mathbf{1}[\mathrm{top}_A(i)=a]\,\mathbf{1}[\mathrm{top}_B(i)=b],$$

implemented as GATEs via Definition B.4. Here and below we fix injective integer encodings $\iota_Q : Q \hookrightarrow \mathbb{Z}$ and $\iota_\Gamma : \Gamma \hookrightarrow \mathbb{Z}$ and identify states and symbols with their codes wherever they appear as residual values or as coefficients. The next-token attributes are then linear combinations

$$q(i+1) \;=\; \sum_{(q,a,b)} \chi_{q,a,b}(i)\cdot q'(q,a,b), \qquad \delta_X(i+1) \;=\; \sum_{(q,a,b)} \chi_{q,a,b}(i)\cdot \delta'_X(q,a,b),$$

and similarly for $u_X(i+1)$, with coefficients $q'(q,a,b), \delta'_X(q,a,b), u'_X(q,a,b)$ read off from δ. The total number of GATE nodes used by the transition logic is $O(|Q|\,|\Gamma|^2)$, a constant.

Output and induction. Give token σ the score $\sum_{(q,a,b):\,\delta(q,a,b)\text{ emits }\sigma} \chi_{q,a,b}(i)$. Exactly one indicator equals one, so the correct next token has score one and every other token has score zero. Its attributes commit the chosen transition at the next position. The stack invariant and transition table therefore prove the simulation by induction, with one token per machine step. Accepting states are recognized from the token's control-state attribute. □

Uniform interpreter form. Fix a universal Turing machine U with constant state set and constant tape alphabet, viewed as a fixed two-stack machine under the standard effective simulation. Applying the construction above to this single U gives a single ALM $\mathcal{A}_U$, with fixed token alphabet Σ_U, fixed transition logic, and $r = 4$ memories. For any Turing machine M and input x, prepend an effectively computable preamble that encodes $\langle M, x\rangle$ as the input to U. The run of $\mathcal{A}_U$ then simulates $U(\langle M, x\rangle)$, and hence $M(x)$. M changes only the input prefix, not the ALM alphabet, memories, gates, or transition logic.

Remark B.9 (Encoding the input). The reduction above describes one step of M in terms of one emitted ALM token. To simulate M on a specific input x, prepend a preamble of push-only tokens that places x onto stack A (or any designated input stack); from the position immediately after the preamble, the construction faithfully simulates M's computation on x. For an input of length n and T simulated steps, the input prefix has $O(n)$ tokens and the generated computation has T tokens, giving $N = O(n+T)$ tokens in total.

B.5 Proof of Proposition 3.5: ALMs Simulate Word-RAM Machines

We use the usual word-RAM conventions: registers and memory cells hold w-bit integers, and arithmetic, comparisons, indexed memory access, and branching have unit cost.

Word arithmetic. Represent each word by its w bits; its integer value is the linear combination $\sum_{k=0}^{w-1} 2^k b_k$. The usual word operations have elementary polynomial-size Boolean circuits: addition and subtraction use carry or borrow chains, multiplication uses shifted partial sums, and binary long division uses w stages of comparison and subtraction. Bitwise operations act on individual bits, and shifts use multiplexers. Reduction modulo 2^w keeps the low w bits. On Boolean inputs, $\mathrm{NOT}(b) = 1-b$ and $\mathrm{AND}(a,b) = \mathrm{ReLU}(a+b-1)$, so these circuits are realized by linear maps and ReGLU gates. Instruction decoding uses the same Boolean operations. The ALM and 2D

transformer are fixed for each word size w: their circuit depth and width may depend polynomially on w, while the number of memories remains $O(1)$. The entire word circuit is evaluated within one decoding position.

An additional memory $\mathcal{M}_{\mathrm{code}}$ supplies instruction fetch. When instructions and data occupy the same RAM memory, it is a copy of $\mathcal{M}_{\mathrm{main}}$, receiving the same input-prefix writes and deferred stores. A fixed number of instruction words uses a fixed number of such read ports. Each stored word includes its bit vector, so fetching a word also supplies the inputs to the arithmetic circuit.

Input prefix. The supplied prefix writes the program and nonzero initial data into the memory copies and the initial register values into $\mathcal{M}_{\mathrm{reg}}$. Unwritten cells are zero. A start token carries the initial program counter and no deferred write. Instruction execution begins after these loading writes.

Simulation, large-alphabet regime. We use a register memory $\mathcal{M}_{\mathrm{reg}}$ storing the register file and a main memory $\mathcal{M}_{\mathrm{main}}$ mirroring Mem. As in the Turing-machine simulation, the simulation is autoregressive: at each position the emitted token *carries the action* to be applied at the next position, and the next position both materializes that action into memory and decides the action that follows.

Each emitted token carries the carried state $(\mathrm{PC},\ \mathrm{dst},\ \mathrm{val},\ \mathrm{st})$, where $\mathrm{PC}, \mathrm{val} \in \{0, \ldots, 2^w - 1\}$, dst indexes the constant register file or denotes no register write, and st is the deferred main-memory store (an address–value pair plus a store/no-store flag). The deferral is forced by the ALM's write-before-read protocol: at position i the instruction is fetched and its operands are read *after* the memories' write steps, so neither the register update nor the store produced by PC_i's instruction can be committed at position i itself; both are carried forward in the token and materialized at position $i + 1$, just as the TM proof carries the push symbol $u_X \in \Gamma$ in the token and materializes it into $\mathcal{S}_X[D_X]$ at the next position. The carried state has $O(w)$ bits, so the alphabet has size $|\Sigma| = 2^{O(w)}$; the embedding unpacks each token into its fields and their bits, so all carried fields are available at layer 0.

At position i, the register cycle commits the update carried by the current token and reads the constant-size register file. A lookup at PC_i retrieves the instruction from $\mathcal{M}_{\mathrm{code}}$ after committing the deferred store to that copy. The main-memory cycle commits the same deferred store st_i and reads the address determined by the instruction and register values. These dependencies determine the memory order. The operation and register indices range over fixed finite sets, so equality indicators and linear combinations select the required operation and operands, coordinate by coordinate. Constants and branch targets are read from the instruction. The word circuit then performs the instruction's arithmetic and forms the next token: PC_{i+1} from the (possibly conditional) branch, and the new deferred updates $(\mathrm{dst}_{i+1}, \mathrm{val}_{i+1}, \mathrm{st}_{i+1})$ from this position's instruction. All resulting words are available both as bits and as their packed integer values.

Inductively, after the current token's deferred writes, the register file, memory copies, and PC_i equal the corresponding RAM configuration. The fetch and arithmetic above therefore compute exactly its next transition, whose effects are carried by the next token. The start token establishes the base case from the loaded image. Thus the ALM's own emitted continuation simulates the RAM execution.

Composing with Theorem 3.2, a 2D transformer simulates the word-RAM with one position per RAM step and $\mathrm{poly}(w)$ gates. The output scores have the quadratic-penalty form of Appendix C.2. Since the computed fields already form a valid token, that token uniquely maximizes the score; returning its attribute tuple evaluates the argmax without enumerating Σ. Its embedding is evaluated by writing the fields and their bits into the residual stream. These evaluations take $\mathrm{poly}(w)$ operations and agree with the embedding and output matrices of the compiled 2D transformer.

$O(w)$-alphabet regime via CumSum. The $O(w)$-bit carried state can also be transmitted across the position boundary using an $O(w)$-size alphabet, at the cost of $O(w)$ ALM positions per RAM step. Write it as $d = O(1)$ fields $x_f \in \{0, \ldots, 2^m - 1\}$, padding each field to the same width $m = O(w)$, and put $M = dm$. Use the alphabet

$$\Sigma \;=\; \left\{0,\ \pm 2^0,\ \pm 2^1,\ \ldots,\ \pm 2^{m-1}\right\} \;\cup\; \Sigma_{\mathrm{ctrl}},$$

where Σ_{ctrl} contains a constant number of markers. A cycle consists of an encode-start marker E, M encode symbols, a firing marker F, and M clear symbols. A masked LOOKUP at a constant key stores the most recent marker and its position. Subtracting that position from **pos** gives the phase

offset; d fixed intervals of length m identify the active field using $O(1)$ threshold gates. Each field has a dedicated vector-valued CUMSUM: a symbol $\pm 2^k$ contributes ± 1 to coordinate k and zero to the other coordinates. These token attributes are supplied by the embedding. Write the coordinates as $U_{f,k}$ and their weighted sum as $S_f = \sum_k 2^k U_{f,k}$. Other fields receive zero.

At F, all fields of the current carried tuple are reconstructed by

$$S_f = \sum_{k=0}^{m-1} (2b_{f,k} - 1)2^k = 2x_f - (2^m - 1), \qquad x_f = \frac{S_f + 2^m - 1}{2}.$$

The individual bits are also available as $b_{f,k} = (U_{f,k} + 1)/2$, providing the inputs to the word-arithmetic circuit without extracting bits from a packed integer. The RAM memories then commit the tuple's deferred writes and compute the next carried tuple by the large-alphabet circuit. Later in the same gate graph, one masked LOOKUP per field stores that next tuple under a constant key, with writes enabled only at F. These staged fields are re-read during the next encode phase. Outside firing or loading markers, all RAM writes are disabled and RAM lookup keys and queries are set to zero. At the markers, the reconstruction identity guarantees that these fields are integers.

The serializer is prefix-driven as follows. Just before emitting the next bit of field f, let its weight be $P = 2^k$. The weights already emitted for that field sum to $A = 2^m - 2P$, so the remainder is recovered from the current prefix sum and the staged field:

$$r = x_f - \frac{S_f + A}{2}, \qquad b = \mathrm{step}(r - P), \qquad s = (2b - 1)P.$$

The invariant $0 \le r < 2P$ holds initially and is preserved because the next remainder is $r - bP \in [0, P)$. The quantity $(S_f + A)/2$ is the sum of already emitted one-bit weights, so r is integral. Inactive comparison inputs are set to zero. The next token is s. For the first bit of each field, $P = 2^{m-1}$ is a fixed constant; otherwise P is half the magnitude of the current encode symbol. Thus P and r are recomputed from the prefix at each position; no residual coordinate is carried between positions. After the last bit of the last field, the next token is F.

For clearing, a separate LOOKUP stores the M encode symbols under the reusable keys $0, \ldots, M-1$. If the encode-start marker is at position E, the symbol at position $E + 1 + a$ is written at key a. Writes are masked off outside the encode phase, with the inactive key set to zero. If the firing marker is at position F, then at position $F + c$, $0 \le c < M$, query key c and emit the negative of that stored symbol; outside clearing, the query is zero. Before F, every key has been overwritten by its symbol from the current cycle, and no replay-memory writes occur during clearing. Latest-write lookup therefore returns exactly the current cycle's encode symbols. The clear symbols cancel them field by field, returning every $U_{f,k}$ and S_f to zero. The last clear symbol emits E, beginning serialization of the staged next tuple.

The supplied loading prefix uses the same field encoding and a separate loading marker, treated as F for phase routing, that enables deferred writes without executing an instruction; its clear blocks also reset the sums. It ends by encoding the initial PC with no deferred writes and supplying F, which computes and stages the first RAM transition. Loading each tuple costs $O(w)$ positions in this regime. Induction over cycles now establishes both the encode/clear invariant and the RAM simulation invariant at each firing position. All reads depend on earlier nodes in the order phase lookup, field sums, RAM memories, staged fields, and next-symbol logic. Each cycle uses $2dm + 2 = O(w)$ positions and $O(1)$ memories and $\mathrm{poly}(w)$ gates per position. The computed next symbol belongs to Σ, so the structured embedding and output convention above also applies here. This proves the small-alphabet construction for Proposition 3.5.

Input and generated positions. For n input words and P program words, the supplied prefix uses $O(n+P)$ positions in the large-alphabet regime and $O(w(n+P))$ in the small-alphabet regime. The continuation for T RAM steps uses T and $O(wT)$ positions, respectively. Evaluating the supplied prefix contributes its own cost when measuring the full input and continuation; these positions are not extra steps of the simulated RAM.

B.6 Finite-Horizon Softmax/SwiGLU Bridge

This appendix transfers the exact construction to softmax/SwiGLU on a finite run with positive lookup and output margins. We use $x\sigma(\beta x) = \mathrm{SiLU}(\beta x)/\beta$ as the sharpened ReLU; its factor $1/\beta$ is absorbed into the down-projection.

Theorem B.10 (Finite-horizon robustness)**.** *Fix a compiled construction from Theorem 3.2 and a horizon $T \in \mathbb{N}$. Let $\Delta_T > 0$ be the minimum lookup-score margin and $\Gamma_T > 0$ the minimum output-argmax margin in the hardmax/ReGLU execution up to time T, let L be the network depth, $H \geq 1$ an upper bound on the number of heads per layer, and $V_{\max}$ a uniform bound on the intermediate residual ℓ_∞ norms encountered up to time T. Set $B = \max(1, V_{\max})$. There is a polynomial $K(B) \geq 1$, depending only on the compiled construction, that bounds error propagation to every intermediate residual, lookup score, and output logit as specified in the proof below. For any $\tau \in (0, \min(1, \Gamma_T/2, \Delta_T/2)]$, the thresholds*

$$S^* = \frac{2}{\Delta_T} \log\left(\frac{8H(T+1)BK(B)}{\tau}\right), \qquad \beta^* = \frac{4HBK(B)}{\tau}$$

suffice: for every $S \geq S^*$ and $\beta \geq \beta^*$, the softmax/SwiGLU transformer with attention scale S and SwiGLU sharpness β emits exactly the same token sequence as the hardmax/ReGLU construction at every position $0, 1, \ldots, T$. In particular $S^* = O(\Delta_T^{-1} \log(2H(T+1)B/\tau))$ and β^* is polynomial in the inverse tolerance and the residual bound, with constants depending on the construction.

Lemma B.11 (Softmax concentrates on a strict maximizer)**.** *Let $s_{j^\star}$ exceed every other score by at least $\Delta > 0$. Then for attention scale $S \geq 0$,*

$$\mathrm{softmax}_{j^\star}(Ss) \geq 1 - (n-1)e^{-S\Delta},$$

where n is the number of candidates.

Proof. This is the standard bound

$$\frac{e^{Ss_{j^\star}}}{\sum_j e^{Ss_j}} = \frac{1}{1 + \sum_{j \neq j^\star} e^{S(s_j - s_{j^\star})}} \geq \frac{1}{1 + (n-1)e^{-S\Delta}} \geq 1 - (n-1)e^{-S\Delta}.$$

□

Proof of Theorem B.10. Use the maximum coordinate norm over all positions through T, and split the forward pass into D affine, attention, and gated substeps. On the unit neighborhood of the exact residuals, affine maps have fixed operator norms, and gated products and unscaled lookup scores have Lipschitz constants $O(B)$. Uniform averaging and selection of a fixed position have norm at most one. Choose a construction-dependent $C \geq 2$ so that $A = CB^2$ bounds these constants and the projection factors.

Assume earlier substeps have preserved each lookup maximizer with gap at least $\Delta_T/2$. Lemma B.11 bounds the error between its softmax read and selection of that position by $2C(T+1)Be^{-S\Delta_T/2}$. For gates,

$$|x\sigma(\beta x) - \mathrm{ReLU}(x)| = \frac{|x|}{1 + e^{\beta|x|}} \leq \frac{1}{e\beta},$$

so multiplying by the other gate input adds at most $CB/(e\beta)$. Uniform heads need no sharpening. Allowing an additional per-head attention error $\varepsilon_{\mathrm{attn}}$, set

$$\eta = 2H(T+1)Be^{-S\Delta_T/2} + \frac{HB}{\beta} + H\varepsilon_{\mathrm{attn}}.$$

After increasing C if needed, the substep errors satisfy $e_0 = 0$ and $e_{d+1} \leq A(e_d + \eta)$. Thus $e_d \leq \eta \sum_{j=1}^{d} A^j$, and $K(B) = A^{D+2}$ bounds the error in every residual, unscaled lookup score, and output logit by $K(B)\eta$.

For this theorem $\varepsilon_{\mathrm{attn}} = 0$. The stated thresholds make the two remaining terms of η at most $\tau/(4K(B))$ each. All errors are therefore at most $\tau/2$. This closes the induction: residuals stay in the unit neighborhood, and lookup gaps remain at least $\Delta_T - \tau \geq \Delta_T/2$. The output gap is at least $\Gamma_T - \tau \geq \Gamma_T/2 > 0$. These comparisons use identical token prefixes; starting from the same prompt, induction over generated positions supplies precisely that premise and proves equality of the token sequences. □

B.6.1 Composition with the Softmax Data Structure

From head accuracy to network accuracy. For a fixed softmax network and a prescribed finite run, all exact intermediate activations lie in a bounded set. Affine maps, ReGLU/SwiGLU gates, and value-weighted softmax attention are locally Lipschitz, so their Lipschitz constants have a finite bound on a closed bounded neighborhood of these activations. Let $C_T \geq 1$ bound the layer constants there, using the maximum coordinate norm over all prefix positions. If the error introduced at each layer is at most δ, then $e_0 = 0$ and $e_{\ell+1} \leq C_T e_\ell + \delta$ give

$$e_L \leq \delta \sum_{j=0}^{L-1} C_T^j \leq L C_T^L \delta.$$

Choose $\delta = \min(\varepsilon, 1)/(LC_T^L)$ for $L \geq 1$; the same bound at every earlier layer keeps the approximate activations in a unit neighborhood and closes the induction. Choose each head's tolerance small enough to meet this layer budget after concatenation, output projection, and the feed-forward block. Their fixed operator and local Lipschitz bounds supply the required factors. The value bound in Theorem 2.1 covers the same neighborhood. Thus C_T is a bound derived from this finite run, not an additional global regularity assumption. The attention algorithm's logarithmic accuracy term gains $\log L + L \log C_T$, together with the fixed projection factors. This argument concerns numerical approximation of a softmax network; the exact hardmax simulation uses no such error budget.

Corollary B.12 (Approximate softmax/SwiGLU simulation, finite horizon)**.** *Fix a compiled ALM, a horizon T covering $N = T + 1$ prefix positions, and a hardmax/ReGLU trace with minimum lookup and output-argmax margins at least $\gamma_T > 0$. Let $B = \max(1, V_{\max})$ bound its intermediate residual norms as in Theorem B.10, let $K(B)$ be the polynomial from that theorem, and put $\hat{\gamma}_T = \min(1, \gamma_T)$. Choose S, β by that theorem with $\tau = \hat{\gamma}_T/4$ and choose*

$$\varepsilon_{\mathrm{attn}} = \frac{\hat{\gamma}_T}{16HK(B)}.$$

Replacing each lookup head's softmax KV cache by the online approximate 2D softmax data structure of Theorem 2.1 run at tolerance $\varepsilon_{\mathrm{attn}}$, the resulting softmax/SwiGLU 2D transformer reproduces the hardmax/ReGLU trace exactly through time T. The per-token cost is $\mathrm{polylog}(N, B, 1/\hat{\gamma}_T)$ in the real-RAM model, with constants depending on the compiled construction.

Proof. In the error recurrence of Theorem B.10, the two sharpening errors contribute at most $\tau/(2K(B))$ to η, and the data-structure error contributes at most $\tau/(4K(B))$. Thus every residual, lookup score, and output logit differs from its ideal value by at most $3\tau/4 < \hat{\gamma}_T/4$. The same layer induction preserves lookup decisions; induction over tokens preserves the output sequence. Each layer's cache stores keys and values computed from the 2D transformer's actual prefix, so the data-structure guarantee applies to those perturbed inputs at every step. Their value bound is $O(B)$ in the maintained unit neighbourhood. Theorem 2.1 therefore gives cost $\mathrm{polylog}(NB/\varepsilon_{\mathrm{attn}})$ per head. Since H is fixed and $K(B)$ is polynomial in B, this is the stated bound. Aggregate heads use exact running sums and counts. □

C Turning Programs to Transformer Weights: Full Details

Theorems 3.2 and 3.4 establish that a 2D transformer simulates any Turing computation in principle; this appendix develops the concrete compiler that realizes the construction. §C.1 gives the formal semantics of CALM, §C.2 the back-end (scheduling MILP, slot reuse via interval coloring, cancel masks, head packing, output layer, and the first Futamura projection), §C.3 the WebAssembly subset we target, and §C.4 the multi-path verification methodology that exercises the full pipeline against independent references.

C.1 The CALM Language

This appendix specifies CALM's primitives and proves the correspondence with the ALM.

C.1.1 Overview

A CALM program maps each prefix $w_0, \ldots, w_i$ to the next token w_{i+1} using the memory and compute primitives of §4.1. Expanding these primitives gives the gate graph of Definition B.3.

C.1.2 Types and Values

CALM has three primitive types:

- `Token`: a member of the finite alphabet Σ.
- `Int`: an unbounded integer. The WASM interpreter represents 32-bit words by bytes and implements wraparound arithmetic using these integer operations.
- `Q`: an exact rational, used as a compile-time coefficient and for intermediate values inside the CUMSUM implementation.

A `Sequence[T]` is a value that holds a `T` at every prefix position. Every CALM expression has a sequence type; the runtime evaluates it at the current position, and the compiler materializes it into a residual-stream slot.

C.1.3 Primitive Semantics

We give the formal semantics of each primitive introduced in §4.1. Throughout, q, k, v are integer-valued expressions in already-computed values, and $c_j \in \mathbb{Q}$ are compile-time rational coefficients.

Memory primitives. Each call site of a memory primitive owns a private memory $\mathcal{M}$. Memory access is atomic: there is no standalone read or write, only the two paired primitives below, evaluated at the current position i.

- LOOKUP(q, k, v): simultaneously sets $\mathcal{M}[k] := v$ and returns $\mathcal{M}[q]$, the latest value written at integer query q — the current position's own write included (write-before-read) — or zero if no visible position $j \leq i$ wrote at that index. An optional mask restricts writes to visible positions. Lemma B.6 gives the exact implementation, including the default-zero case.
- CUMSUM(v): writes v at the current position and returns $\sum_{t \leq i} v(t)$. Realized as one uniform-attention head ($W_Q = W_K = 0$) followed by a GATE that multiplies the prefix average by $\mathbf{pos} + \mathbf{1}$.

Compute primitives. The constant-depth circuit between memories is built from two primitives, with the standard arithmetic vocabulary derived mechanically:

- Linear combination $c_1 x_1 + \cdots + c_k x_k$ of earlier values, with constant rational coefficients $c_j \in \mathbb{Q}$, realized as residual-stream wiring with no nonlinear gate.
- GATE$(a, b) = a\ \mathrm{ReLU}(b)$, realized as one ReGLU neuron in the feed-forward block.

Step indicators, equality, selection, and multiplication are the constant-size derived circuits of Definition B.4. The artifact uses ReGLU gates; the softmax/SwiGLU approximation is analyzed in Appendix B.6.

C.1.4 Token Output

A CALM program terminates by declaring a finite collection of *output scoring expressions*, one per candidate next-token type. The runtime emits the token whose score is largest, using the tie rule of Appendix B.2. In the artifact the scoring expressions are quadratic penalties of the form $C \cdot \mathrm{type}(i) - \sum_k (p_k(i) - \hat{v}_k)^2$ for a positive integer constant C. When the computed integer attributes identify a valid token uniquely, its score is C and every other token scores at most $C - 1$. This supplies the positive margin for the bounded-horizon robustness of Theorem B.10.

C.1.5 The Gate Graph

Each LOOKUP becomes a lookup node; each CUMSUM becomes an AGGREGATE followed by multiplication by $\mathbf{pos} + \mathbf{1}$; and each compute primitive becomes linear wiring or GATE nodes. Appendix C.2 schedules this graph and supplies the positional features needed by its attention heads.

C.1.6 Soundness and Completeness

Compilation soundness. Theorem 4.1 follows from Theorem 3.2: the compilation pipeline of Appendix C.2 preserves the residual-stream invariant by construction, so the gate-graph evaluation and the transformer forward pass agree token by token. For softmax/SwiGLU, Theorem B.10 gives the finite-run guarantee under its stated margin conditions.

Theorem C.1 (CALM completeness for the ALM)**.** *Every Append-only Lookup Machine is expressible in surface CALM with the primitive semantics above.*

Proof. For each ALM memory, translate every lookup read to a CALM call with that read's query and the memory's shared write expressions. All such private copies receive identical writes, so they return the same values as the source memory. Translate each cumulative-sum read to $\textsc{CumSum}(v)$ with the same written values. Translate the intervening circuit using linear combinations and GATE, and retain the same finite output scoring family and tie rule. Induction over the memory order proves equality at each position; induction over positions proves equality of the token streams. □

C.2 Compilation Pipeline: Scheduling, Slot Reuse, and Specialization

The compiler schedules the gate graph into attention and FFN phases, reuses residual columns after their last read, and writes the resulting weight matrices analytically.

C.2.1 Per-Layer Phase Structure

Each transformer layer exposes two phases that produce residual-stream values, in a fixed order:

(1) **Attention.** LOOKUP and AGGREGATE gates read from the prefix; their outputs are written into residual-stream slots.

(2) **FFN.** GATE gates and their derived operations (step, equality, selection, integer multiplication) fire; their outputs are written into residual-stream slots.

A gate may be assigned to phase (1) only if it is a LOOKUP or AGGREGATE, and to phase (2) only if it is a GATE. The output projections write the results into their assigned columns.

Lookup pre-materialization. Before a LOOKUP attention phase fires, its K and V projections require $k(j)^2$ and $\mu(j)v(j)$, respectively, as linear functions of the residual. The compiler therefore inserts, into the gate graph, the GATE nodes computing k^2 and each coordinate of μv, scheduled in the FFN phase of a strictly earlier layer; the MILP precedence constraint enforces this automatically, since within a layer the FFN phase follows the attention phase. The selected-key equality test and final product are scheduled after the attention phase, as in Lemma B.6. All these helper nodes participate in scheduling and slot allocation.

C.2.2 The Scheduling MILP

Let $G = (V, E)$ be the gate graph and L the layer budget. For each gate $g \in V$ the scheduler selects a layer $\ell(g) \in \{0, \ldots, L-1\}$ and a phase $p(g) \in \{1, 2\}$ subject to:

- **Type compatibility.** $p(g) = 1$ if g is a LOOKUP or AGGREGATE; $p(g) = 2$ if g is a GATE.
- **Precedence.** For every edge $u \to v$ in E, u is assigned strictly before v in the lexicographic ordering on (ℓ, p) tuples.

Each gate's value occupies a residual-stream slot from the end of the phase that produced it until its last reader. At each phase boundary the number of *live* gates (those whose slot is not yet free) is the dependency width at that boundary. Let D be the maximum dependency width across all boundaries; the objective is to minimize D, the number of *reusable* slots required after slot reuse. All temporary values, including lookup squares and masked values, participate in this liveness calculation. Permanent input and positional columns are counted separately. Head packing and cancellation may require padding to an even residual width.

A per-boundary liveness indicator linearizes D, and the entire program is mixed-integer linear. The implementation uses PuLP/HiGHS; if L is too small, the solver reports infeasibility, in which case the compiler doubles the layer budget and retries. Compilation is total: a schedule always exists at a sufficiently large budget (at worst one node per phase), so the loop terminates.

C.2.3 Slot Reuse via Interval Coloring

Once the schedule is fixed, every gate has a known lifetime: an interval from the end of its producing phase to its last reader. Two gates can share a residual-stream slot iff their intervals do not overlap. Computing the smallest number of slots is interval graph coloring, solved greedily in $O(|V| \log |V|)$ time on intervals sorted by start time: walk the sorted list and assign each gate the lowest-numbered free slot. The chromatic number equals the dependency width D from the MILP, so the number of reusable slots is D, before permanent columns and head-packing constraints are counted.

C.2.4 Slot Reuse and Residual Addition

Reusing a slot containing x for a new value y requires adding $y - x$. In an attention phase, a self-selecting head with query $(i, 1)$ and key $(2j, -j^2)$ returns the current position's old value; its output projection subtracts that value. In an FFN phase, the ReGLU neuron $(-x)\,\mathrm{ReLU}(1)$ supplies the same cancellation. All heads or neurons in a phase read its incoming residual, so last readers see the old value before the simultaneous update. The resulting slot is $x + (y - x) = y$, preserving the residual invariant. The released scheduler accounts for the lifetimes and cancellation resources when allocating slots.

C.2.5 Head Packing

A 2D head carries two scalar values. Reads with the same attention pattern can share these channels: two uniform averages, or two lookups with identical queries, keys, and visibility masks. When both channels carry lookup values, the default-zero test still needs the selected key; an additional head with the same pattern supplies it. This check can be omitted when a visible match always exists. Head counts include these additional channels.

C.2.6 Output Layer

The output projection $W_{\text{out}} \in \mathbb{R}^{|\Sigma| \times d_{\text{model}}}$ evaluates each token's score as a fixed linear functional of materialized residual columns. For the compiled interpreter, the scoring expressions have the quadratic penalty form $C \cdot \mathrm{type}(i) - \sum_k (p_k(i) - \hat{v}_k)^2$ described in Appendix C.1; using materialized p_k^2 columns and the linear identity $-(p_k - \hat{v}_k)^2 = -p_k^2 + 2\hat{v}_k\, p_k - \hat{v}_k^2$, row σ of W_{out} places -1 on each p_k^2 column, $2\hat{v}_k(\sigma)$ on each p_k column, C on the type-indicator column, and $-\sum_k \hat{v}_k(\sigma)^2$ on the always-one column. This uses shared attribute columns rather than one column per token. Whenever the integer attributes identify a valid token uniquely, its score exceeds every alternative by at least one, as required by Appendix B.6.

C.2.7 Specialization: First Futamura Projection

This is the back-end pass referenced in §4.2. Given a WASM program with N_{prog} instructions, the specializer adds $2N_{\text{prog}}$ shared GATE neurons that compute the cursor step indicators $\mathbf{1}[\text{cursor} \geq i]$ for $i = 0, \ldots, N_{\text{prog}} - 1$, and rewrites each fetched instruction field as the linear combination

$$\text{field}(\text{cursor}) \;=\; c_0 \;+\; \sum_{i=1}^{N_{\text{prog}}-1} (c_i - c_{i-1})\, \mathbf{1}[\text{cursor} \geq i],$$

with c_i the field's value at instruction i. We then drop the instruction-fetch LOOKUP head of the universal interpreter; the program prefix vanishes from the input. For every valid instruction index the sum telescopes to the original field value, so instruction fetch and hence the generated trace are unchanged. The same scheduling and slot-reuse machinery handles the rewritten graph.

C.2.8 Implementation Notes

The MILP, slot reuse, cancel-mask emission, and analytic weight construction are implemented in the public repository of Appendix C.4 under `transformer_vm/scheduler/` (MILP, PuLP/HiGHS)

and `transformer_vm/model/` (weight construction and the PyTorch transformer). The CLI entry points are documented in Appendix C.4.

C.3 A Variant of WebAssembly

The compiled executor is a variant of 32-bit integer WebAssembly that preserves the usual control-flow, stack, call, local, and linear-memory structure. The main simplification is numerical: primitive arithmetic is restricted to addition and subtraction, while more complex numeric instructions are compiled away by lowering before transformer compilation.

C.3.1 Instruction Set

The program-prefix instruction set contains 36 opcode tokens. Of these, 33 are WebAssembly-derived and 3 are internal pseudo-operations: `halt`, `output`, and `input_base`. The WebAssembly-derived operations cover:

- control flow: `br`, `br_if`, `call`, `return`;
- stack manipulation: `drop`, `select`;
- locals: `local.get`, `local.set`, `local.tee`;
- globals: `global.get`, `global.set`;
- full-width loads: `i32.load`;
- 8-bit loads: `i32.load8_s`, `i32.load8_u`;
- 16-bit loads: `i32.load16_s`, `i32.load16_u`;
- stores: `i32.store`, `i32.store8`, `i32.store16`;
- arithmetic: `i32.const`, `i32.add`, `i32.sub`;
- comparisons: `i32.eqz`, `i32.eq`, `i32.ne`, `i32.lt_s`, `i32.lt_u`, `i32.gt_s`, `i32.gt_u`, `i32.le_s`, `i32.le_u`, `i32.ge_s`, `i32.ge_u`.

Lowering for complex arithmetic. Multiplication, division, and bitwise instructions are reached by a lowering pass that rewrites them into the primitive subset before compilation. We omit these instructions from the primitive executor to keep the analytic construction simple, not because they are incompatible with the 2D-head framework or the ALM.

Scope of the executor. The executor is a precisely-scoped artifact, not a full C/WASM implementation. Table 2 summarizes what is supported natively by the primitive opcode set, what is supported via the lowering pass, and what is outside the scope of the artifact.

Table 2: Scope of the WASM-subset executor compiled by CALM.

Supported (native)	integer control flow (`br`, `br_if`, `call`, `return`); locals/globals; value-stack manipulation; byte-addressed linear-memory loads and stores at 8/16/32-bit widths; integer addition, subtraction, and comparisons; bounded recursion within trace and memory limits; 32-bit deterministic semantics.
Supported (lowered)	multiplication, division, and bitwise operations are rewritten into the primitive opcode set by a pre-compilation pass before the gate graph is built.
Not supported	floating-point arithmetic; OS interaction (system calls, IO, file system); concurrency or threads; arbitrary C library calls; nondeterminism; programs that exceed the configured trace length, memory size, or call-stack depth; full WASM validation semantics for nondeterministic inputs.

In particular, claims that the compiler "runs C programs" refer throughout to programs that fall in the supported (native or lowered) rows of Table 2.

C.3.2 Machine State

The value stack, call stack, local and global storage, and byte-addressable linear memory all live in the ALM trace: running counters such as stack depth and instruction pointer come from AGGREGATE nodes, while random-access retrieval is expressed through LOOKUP nodes over the append-only trace.

Stacks. The value stack and call stack both use the push-filtered-key idea from Appendix B.4. Each push writes its payload under the current depth, and later lookups query the current depth (or a nearby depth such as depth minus one) to recover the top entries.

Locals and memory. Local variables are addressed by a composite key combining call depth, local index, and byte offset. Linear memory uses write-address lookups: the machine retrieves both the candidate byte and the address from which it came, then verifies equality with the queried address before accepting the byte. This is the exact-default-zero pattern discussed in Lemma B.6.

C.3.3 Two Structural Ideas

Circle-point opcode dispatch. To dispatch over many opcodes without a logarithmic-depth binary dispatch circuit, the construction assigns each opcode a distinct integer point on a common circle in $\mathbb{Z}^2$. The fetched opcode carries those coordinates in the program prefix. For a target opcode with point (p_x, p_y), one GATE evaluates

$$\mathrm{ReLU}(p_x x + p_y y - R^2 + 1).$$

The argument equals 1 on the matching opcode. For any other integer point on the circle, strict Cauchy–Schwarz and integrality give $p_x x + p_y y \leq R^2 - 1$, so the output is zero.

Byte-level arithmetic. The machine works bytewise. A 32-bit operation is decomposed into four byte steps with carry or borrow propagation carried through the trace. This keeps the nonlinear primitives simple: comparisons reduce to step functions and equality tests, while addition and subtraction reduce to byte arithmetic plus a small amount of control logic.

C.3.4 Compiling the Executor

C or WebAssembly input is lowered to the primitive instruction stream above, the resulting gate graph is scheduled and slot-reused by Appendix C.2, and the final weight matrices are written directly into the simplified transformer of Appendix B.1.

C.4 Verification and Reproducibility

The compiler and examples are publicly available at `https://github.com/Percepta-Core/transformer-vm`. The repository's README gives installation requirements and commands for building weights, generating reference traces, and running the universal and specialized models.

Reference checks. `wasm-reference` executes the WASM program directly to generate reference token traces. `wasm-eval` evaluates the CALM graph without transformer weights; its `-nohull` option uses brute-force lookup. `wasm-run` executes the analytically constructed transformer and checks its generated tokens against the reference trace. It uses the C++ inference engine by default; `-python` selects PyTorch. Comparing these paths checks the graph formulas, scheduling, weight construction, and hull cache on the example programs. These implementation tests complement the mathematical proofs; they do not replace them.

Reproducing the construction. After installing the documented dependencies, `uv sync` and `uv run wasm-run` compile the examples, construct the weights, and execute the programs. The `wasm-build` command builds the universal model explicitly. `wasm-specialize` embeds one program into the weights; the README gives a complete example of building and running a specialized model. Program tokens and reference traces are generated under `transformer_vm/data/`. The source includes the compiler and tests; generated weights and traces can be rebuilt with these commands.

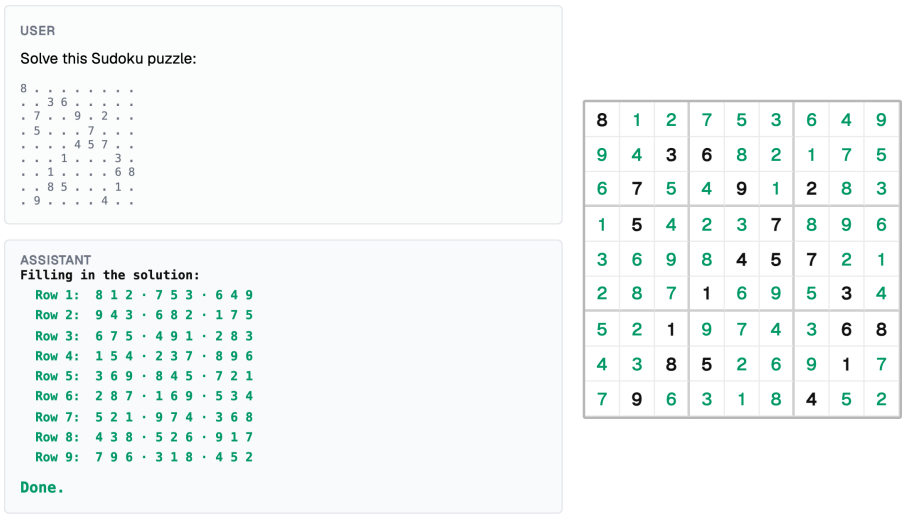


Figure 2: The compiled executor running a backtracking Sudoku solver on a published "hardest Sudoku" instance, end-to-end in a single autoregressive PyTorch run. The trace is verified token by token against the reference interpreter (Appendix C.4).

D Extended Related Work

This appendix expands the comparisons in §6 on attention algorithms, universality, and compilation into weights.

D.1 Subquadratic and Fine-Grained Complexity of Attention

The fine-grained complexity backbone. Duman Keles et al. (2023) were the first to apply fine-grained complexity to attention, reducing from the Orthogonal-Vectors Hypothesis (OVH, implied by SETH) to show that no strongly subquadratic ($N^{2-\Omega(1)}$) algorithm computes the full attention output even with additive $1/\mathrm{poly}(N)$ error, when $d = \omega(\log N)$. Alman and Song (2023) sharpened this into a tight upper-and-lower-bound dichotomy parameterized by the entry bound $B = \max(\|Q\|_\infty, \|K\|_\infty, \|V\|_\infty)$:

- Upper bound: when $B = o(\sqrt{\log N})$ and $d = O(\log N)$, there is an $N^{1+o(1)}$ algorithm via a polynomial-method approximation of exp followed by fast rectangular matrix multiplication.
- Lower bound: at dimension $d = \Theta(\log N)$ and $B = \Theta(\sqrt{\log N})$, no $N^{2-\Omega(1)}$ algorithm exists assuming SETH.

Recent work on arbitrary entries. Gupta et al. (2026) treat arbitrary entries and any constant dimension d, achieving $\widetilde{O}(N^{2-1/d})$ time for the full attention output, matching a lower bound for $d = 2^{\Theta(\log^* N)}$. At $d = 2$ this gives $\widetilde{O}(N^{3/2})$. Our Theorem 2.1 is the first online algorithm for 2D attention at arbitrary key/query norms, serving every insertion and every query in amortized polylogarithmic time per token.

Kernel density estimation and partition functions. Kernel-density-estimation methods (Charikar and Siminelakis, 2017; Backurs et al., 2021) estimate partition functions or kernel-matrix-vector products. Spring and Shrivastava (2017) give an unbiased LSH-based partition-function estimator; Backurs et al. (2021) extend to more general kernel-matrix algebra. KDEformer (Zandieh et al., 2023) also approximates the full normalized attention output, with spectral-norm guarantees. Our guarantee is deterministic additive ℓ_∞ error for each online 2D-attention query, with arbitrary key/query norms and polylogarithmic amortized cost.

D.2 Computational Power of Transformers

Foundational Turing-completeness results. Siegelmann and Sontag (1995) establish that recurrent neural networks with rational weights and unbounded internal precision can simulate Turing machines; this result concerns recurrent networks rather than transformers. Pérez et al. (2021) prove Turing completeness for transformers with a separate input encoder, hard attention, the non-standard piecewise-linear scoring rule $\mathrm{score}(q,k) = -|\langle q,k\rangle|$, a custom rational positional encoding $\mathrm{PE}(i) = (1, i, 1/i, 1/i^2)$, and unbounded rational precision. The construction recovers the head position $c^{(i)} = \sum_{j<i} m^{(j)}$ by averaging $m^{(j)}$ over the entire move history with self-attention; this average can be maintained with a running sum. Exact evaluation of their $1/i$, $1/i^2$ positional encoding uses rational arithmetic. Bhattamishra et al. (2020) give an alternative hard-attention construction by simulating a recurrent network, again with arbitrary precision. Our exact construction in Theorem 3.4 uses integer-keyed memory, with $O(\log(n+T))$-bit arithmetic sufficing for T simulated Turing-machine steps on an input of length n (Appendix B.3); it is a hardmax/ReGLU 2D transformer, with the softmax/SwiGLU bridge holding on any prescribed finite run (Remark 3.3).

Recurrent and looped variants. Looped Transformers (Giannou et al., 2023) compile SUBLEQ into a 13-layer transformer placed inside an external loop; the loop is not part of the transformer architecture, and the softmax is driven into a saturated hardmax regime through large logits. Our exact construction is a single non-looped causal 2D transformer of fixed depth and no external memory, with the hardmax/uniform attention semantics stated in Appendix B.1.

Limitations and chain-of-thought characterizations. Merrill and Sabharwal (2023) place log-precision transformers without CoT in logspace-uniform TC^0. Merrill and Sabharwal (2024) show that polynomially many CoT tokens suffice to recognize all of P, using saturated attention, strict causal masking, and projected pre-norm in their construction; logarithmically many CoT tokens remain simulable in logarithmic space. Nowak et al. (2024) show that recurrent and transformer language models with CoT reasoning represent the same family of distributions over strings as probabilistic Turing machines, recasting the question in probability rather than classical computability. Li and Wang (2025) prove that a constant bit-size transformer simulates any Turing machine on inputs of arbitrary length, with $\mathsf{SPACE}[s(n)]$ exactly characterizing the expressive power at context length $s(n)$. Roberts (2024) gives a direct Turing-completeness analysis of causal transformers under suitable embedding-sparsity assumptions. Our construction gives explicit weights for indexed memory and control flow, with one token per Turing-machine step and the word-RAM simulation bound of Corollary 3.6.

D.3 Programming and Compiling Transformer Weights

The RASP family. RASP (Weiss et al., 2021) is a descriptive specification language whose primitives (`select`, `aggregate`, position-wise maps) abstract attention heads but include arbitrary Python position-wise functions, which makes RASP programs not in general realizable as fixed-precision weights. Tracr (Lindner et al., 2023) compiles a RASP subset to transformer weights in JAX/Haiku; Tracr models have categorical residual subspaces, no LayerNorm in the algorithmic core, and bucketed piecewise-linear approximations for numerical s-ops. RASP-L (Zhou et al., 2024) uses a hand-restricted RASP subset to predict transformer length-generalization; Transformer Programs (Friedman et al., 2023) train constrained transformers and extract RASP-like programs from learned weights. C-RASP (Yang and Chiang, 2024) compiles a temporal counting logic into softmax transformers but the source language is bounded counting, not general computation.

CALM (Appendix C.1) uses exact integer memory operations and ReGLU circuits, and every CALM program compiles to weights by Theorem 4.1. The resulting 2D transformer runs autoregressively without an external loop. The WASM interpreter makes this construction accessible from C: its supported operations and lowering pass are described in Appendix C.3. The program can be supplied in the prefix or embedded in the weights by the specialization pass of Appendix C.2.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The claims are supported by the attention algorithm, simulation proofs, and compiler experiments in Sections 2–5.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Section 7 discusses numerical precision, the finite-run softmax guarantee, and the experiments' focus on constructed weights.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [Yes]

Justification: The statements give their assumptions, and Appendices A–C provide the proofs.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: Appendix C.4 explains how to build the models and compare their outputs with reference executions.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: The compiler, examples, and tests are public. Appendix C.4 links to the repository and its build instructions.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Section 5 describes the test programs and execution settings; Appendix C.4 gives reproduction instructions. No training is involved.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [No]

   Justification: Token correctness is checked exactly against reference outputs. The reported timing measurements have no error bars.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).

- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [No]

   Justification: Section 5 reports CPU-only execution, single-threaded float64 computation, and runtimes. The CPU model, memory capacity, and total project compute are not reported.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: We follow the NeurIPS Code of Ethics. This work involves no human participants or personal data.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [N/A]

    Justification: We identify no distinct societal risks beyond those of executing the same programs on conventional computers.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.

- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The released compiler introduces no additional risks requiring special release safeguards.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [No]

    Justification: We identify the software and puzzle sources, but do not list every license and version.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.

- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: The repository README documents how to build and run the compiler and examples (Appendix C.4).

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The work involves no crowdsourcing or human participants.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: This work involves no human participants, so IRB approval is not applicable.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.

- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [N/A]

    Justification: The core methods were developed by the authors.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.