# Cortex: The AI-First Programming Language Based on Selective State Spaces

**A Comprehensive Implementation Strategy Using Mamba-2 State Space Duality in Rust**

---

**David Jean Charlot, PhD** Open Interface Engineering, Inc. (openIE) University of California, Santa Barbara (UCSB) david@openie.dev | dcharlot@ucsb.edu

**February 2026 | Version 1.0**

*This work is licensed under CC BY-ND 4.0 | Open Access Research*

---

## Abstract

The computational landscape is currently navigating a pivotal transition, moving from the rigid, deterministic execution of explicit instructions toward probabilistic, intent-driven modeling. For decades, the von Neumann architecture has dictated a separation between storage and processing, a paradigm that traditional programming languages like C++ and Rust have optimized to near perfection. However, the rise of large language models (LLMs) has introduced a new mode of computation—one that is semantic, fluid, and capable of generalizing across unseen inputs. Yet, the dominant architecture driving this revolution, the Transformer, remains fundamentally constrained by its quadratic complexity, $O(L^2)$, rendering it unsuitable for infinite-context environments such as operating system kernels, continuous robotic control loops, or real-time autonomous agents.

This report presents a comprehensive implementation strategy and theoretical analysis for **Cortex**, an AI-first programming language built upon **Rust 1.93.0**. Cortex fundamentally reimagines the relationship between code and intelligence by embedding a **Selective State Space Model (SSM)**—specifically leveraging the architecture of **Mamba-2** and the **State Space Duality (SSD)**—as its primary runtime execution unit. Unlike Python-based workflows where AI models are treated as external black boxes accessed via heavy API calls, Cortex integrates the inference kernel directly into the language's binary, allowing for deterministic state evolution guided by learnt dynamics.

We provide a deep dive into the theoretical lineage of this technology, tracing the roots from the Kalman Filters of the 1960s to the High-order Polynomial Projection Operators (HiPPO) of 2020, culminating in the Selective SSMs of today. Furthermore, we articulate a "zero-dependency" implementation plan that bypasses high-level frameworks like PyTorch or Candle. Instead, we propose a bare-metal tensor library and SSM kernel constructed from scratch using Rust's advanced features—including portable SIMD (std::simd) and inline assembly (asm!)—to bridge the gap between historical control theory and modern hardware acceleration.

**Keywords**: state space models, Mamba, selective scan, AI-first programming, Rust, neural-symbolic computing, SSM, hardware-aware compilation

---

# 1. Theoretical Foundations

The conceptual underpinnings of Cortex are not arguably novel discoveries but rather the algorithmic maturation of principles established over seventy years of research in control theory, signal processing, and approximation theory. To understand why Mamba and SSMs represent the future of sequence modeling, one must first trace the lineage of the **State Space Model (SSM)**, a mathematical framework that links continuous physical systems with discrete digital computation.

## 1.1 The Lineage of State Space Models

The core mathematical object in Cortex is the State Space Model. Its origins are firmly rooted in the mid-20th century, specifically in the domain of optimal control and estimation.

### The Kalman Filter and Control Theory (1960s)

In 1960, Rudolf Kalman introduced the Kalman Filter, a recursive solution to the discrete-data linear filtering problem.1 This algorithm revolutionized control systems engineering, playing a pivotal role in the navigational calculations for the Apollo program and the guidance systems of early missiles. The Kalman Filter models a dynamic system using two primary equations:

1. **State Equation:** $x_{k+1} = Ax_k + Bu_k$
2. **Output Equation:** $y_k = Cx_k + Du_k$

In this formulation, $u(t)$ represents the input signal (e.g., sensor readings), $y(t)$ is the output signal (e.g., motor commands), and $x(t)$ is the **latent state**—a compressed representation of the system's history at time $k$. The matrices $A$, $B$, $C$, and $D$ define the dynamics of the system.

- $A$ **(State Transition Matrix):** Dictates how the current state evolves into the next state.
- $B$ **(Input Matrix):** Determines how new inputs influence the state.
- $C$ **(Output Matrix):** Describes how the latent state translates to an observable output.
- $D$ **(Feedthrough Matrix):** Represents a direct path from input to output, often omitting the state entirely.

In classical control theory, these matrices are typically static (Linear Time-Invariant, or LTI) and low-dimensional. The critical insight for modern Artificial Intelligence is that this exact formulation describes a sequence-to-sequence map. If one can learn high-dimensional matrices $A$, $B$, and $C$ from data, the system can theoretically model complex sequences like language or DNA with theoretically infinite context, constrained only by the capacity of the state vector $x(t)$.1

### Discretization: The Zero-Order Hold (ZOH)

The original SSM equations are defined in continuous time ($t$). However, digital computers—and by extension, the Cortex language—operate in discrete time steps ($k$). To apply continuous-time control theory to digital sequences, the differential equations must be **discretized**.

The standard method employed in modern SSMs like Mamba is the **Zero-Order Hold (ZOH)**.1 The ZOH assumption posits that the continuous input signal $u(t)$ holds a constant value during the sampling interval $\Delta$ (delta). This is physically analogous to how a digital-to-analog converter holds a voltage steady between clock ticks.

Mathematically, this transformation converts the continuous parameters $(A, B)$ into discrete parameters $(\bar{A}, \bar{B})$ dependent on the step size $\Delta$:

$$\bar{A} = \exp(\Delta \cdot A)$$

$$\bar{B} = (\bar{A} - I)A^{-1}B$$

This discretization step is not merely a technicality; it is a profound architectural feature. In classical Recurrent Neural Networks (RNNs) like LSTMs or GRUs, the "step size" is implicit—index $k$ transitions to $k+1$ with no notion of physical time. In discretized SSMs, $\Delta$ is a learnable parameter. This allows the model to learn the "resolution" at which it samples the input data. A small $\Delta$ implies high-resolution sampling (ignoring coarse trends), while a large $\Delta$ implies integrating information over a longer period. This property makes SSMs robust to timescale shifts, a capability absent in Transformers and traditional RNNs.5

### HiPPO: Optimal Memory (2020)

Despite the theoretical elegance of SSMs, early attempts to use them for deep learning failed due to the "vanishing gradient" problem. As the sequence length grew, the influence of early inputs on the current state would decay exponentially, effectively causing the model to suffer from amnesia.

In 2020, Gu et al. formalized a solution with the **HiPPO (High-order Polynomial Projection Operators)** framework.7 HiPPO approaches the memory problem from a function approximation perspective: *Given a continuously growing function $f(t)$ representing the history of inputs, how can we optimally approximate it at every step $k$ using a fixed-size state vector $x(t)$ ?*

The framework demonstrates that by projecting the history onto a basis of **orthogonal polynomials**—specifically Legendre polynomials—one can derive a closed-form solution for the optimal state transition matrix $A$. The resulting "HiPPO Matrix" ensures that the state $x(t)$ minimizes the approximation error of the history $f(t)$ with respect to a specific measure.9 This initialization of $A$ allows the state to retain a faithful compression of inputs from thousands of steps ago, enabling the first **Structured State Space (S4)** models to drastically outperform Transformers on the Long Range Arena benchmarks.10 This mathematical guarantee of long-term memory is what makes SSMs a viable backbone for an infinite-context language like Cortex.

## 1.2 The Innovation of Selectivity: From LTI to LTV

While S4 solved the memory retention problem, it remained a **Linear Time-Invariant (LTI)** system. In S4, the matrices $A$, $B$, and $C$ were learned during training but remained fixed (static) during inference for all time steps. This meant the model processed every token with the exact same dynamics. It effectively treated "noise" tokens (e.g., filler words like "um" or whitespace) with the same importance as "signal" tokens (e.g., names or keywords).

**Mamba (2023)** introduced the breakthrough of **Selectivity**, transitioning the architecture from LTI to **Linear Time-Varying (LTV)** systems.11 In the Mamba architecture:

- The parameters $B$, $C$, and the step size $\Delta$ are no longer static weights.
- They are computed as functions of the current input $x_k$.

$$B_k = s_B(x_k)$$

$$C_k = s_C(x_k)$$

$$\Delta_k = \text{softplus}(s_\Delta(x_k))$$

This input-dependence allows the model to “select” what to store in its state at every millisecond.

- If $\Delta_k$ is computed to be close to 0 for a specific input $x_k$, the state update is suppressed. The model effectively “skips” or “ignores” this input, preserving the existing memory.
- If $\Delta_k$ is large, the new input $x_k$ dominates the update, allowing the model to “reset” or “focus” on new information.

This mechanism is conceptually similar to the gating mechanisms found in LSTMs (forget gates) but is applied mathematically to the discretization of the state space matrices. This selectivity allows Mamba to filter out irrelevant data in long sequences, matching the performance of Transformers on language modeling tasks while maintaining linear inference complexity.13 For Cortex, this means the runtime can dynamically allocate its finite “memory bandwidth” to the most salient parts of the code or data stream it is processing.

---

## 2. State-of-the-Art Landscape Analysis

To effectively design Cortex, we must situate the Mamba architecture within the broader competitive landscape of modern AI models. The current SOTA is defined by the tension between **Training Parallelism** (how fast we can learn) and **Inference Complexity** (how fast and cheap we can generate).

### 2.1 The Complexity Gap: Transformers vs. SSMs

The Transformer architecture, specifically its Attention mechanism, has dominated AI due to its high parallelizability during training. However, it suffers from a fundamental bottleneck during inference: the **KV Cache**.

To generate the next token, a Transformer must attend to *all* previous tokens. This requires storing Key (K) and Value (V) vectors for the entire history. As the sequence length ($L$) grows, this cache grows linearly in memory usage ($O(L)$) and the compute required to process it grows quadratically ($O(L^2)$) or linearly ( $O(L)$) depending on optimization, but memory bandwidth remains the bottleneck.15 For a sequence of 1 million tokens, the KV cache can consume hundreds of gigabytes of VRAM, making inference prohibitively expensive and slow.

In contrast, Mamba (SSM) compresses the entire history into the fixed-size state $x \in \mathbb{R}^N$. This state size (e.g., 2KB or 4KB) does not grow with sequence length.

- **Inference Complexity:** $O(1)$ (Constant time per token).
- **Memory Usage:** Constant.
- **Training:** Parallelizable ($O(L)$) via the **Parallel Associative Scan** algorithm (discussed in Part III).

| Feature | Transformer | RNN (Traditional) | Mamba (SSM) |
|---|---|---|---|
| **Training** | Parallelizable ( $O(L^2)$) | Sequential ($O(L)$) | **Parallelizable ( $O(L)$)** |
| **Inference Speed** | Slows down as context grows | Constant ($O(1)$) | **Constant ($O(1)$)** |
| **Inference Memory** | Grows linearly (KV Cache) | Constant | **Constant ($x \in \mathbb{R}^N$)** |
| **Long Context** | VRAM limited (e.g. 128k) | Lossy (Vanishing Gradient) | **Infinite (theory)** |
| **Selectivity** | High (Attention Matrix) | Low (Static Weights) | **High (Input-Dependent)** |

Table 1: Architectural Comparison showing Mamba's "Best of Both Worlds" profile.16

**Insight for Cortex:** This efficiency profile implies that Cortex can theoretically run continuous, unending loops of inference without running out of memory. This is critical for "agentic" workflows where an AI program might run as a

daemon process for days or weeks.

## 2.2 Mamba-2: State Space Duality (SSD)

The most significant recent advancement relevant to our implementation is **Mamba-2**, which introduces the concept of **State Space Duality (SSD)**.18 The authors of Mamba-2 discovered that structured SSMs are mathematically dual to a form of structured linear attention.

- **The Bottleneck of Mamba-1:** While Mamba-1 is efficient, its core "selective scan" operation is memory-bound. It requires sequentially reading and writing to the GPU's fast SRAM, which underutilizes the GPU's massive array of computing cores designed for Matrix Multiplication (MatMul).
- **The SSD Solution:** By imposing a specific constraint on the $A$ matrix—restricting it to a **scalar-times-identity** structure ($A = aI$)—the recurrent SSM computation can be reformulated as a **block-decomposition matrix multiplication**.

This allows the computation to be split into chunks. Within each chunk, the interaction looks like a Matrix Multiplication, which can be dispatched to **Tensor Cores** (specialized hardware on NVIDIA GPUs). The chunks are then linked together via a simplified scan. This hybrid approach achieves 2-8x faster training speeds compared to Mamba-1 while maintaining the same inference characteristics.19

**Strategic Decision for Cortex:** We must implement the **SSD algorithm (Mamba-2)** for the Cortex kernel. While the Mamba-1 scan is elegant, the SSD algorithm is far more hardware-efficient on modern GPUs, bridging the gap between theory and the specific capabilities of silicon like the H100 or consumer RTX 4090s.18

## 2.3 Hybrid Architectures: The Role of Attention

The industry is also observing a trend toward **Hybrid Architectures**. A prime example is **Jamba**, developed by AI21 Labs, which interleaves Mamba layers with Transformer Attention layers and Mixture-of-Experts (MoE) layers.21

- **Ratio:** Jamba typically uses a ratio of 1:7 (1 Attention layer for every 7 Mamba layers).
- **The Logic:** While SSMs are excellent at compressing history, they can struggle with "needle-in-a-haystack" retrieval tasks (finding a specific fact from 500 pages ago) because the state compression is lossy. The Attention layers act as "synchronization points" or "associative memory lookups" that can access the raw token history if needed.

**Implication for Cortex:** While our primary goal is an SSM-native language, a robust implementation of Cortex should likely support a "Hybrid Block" or an "Attention Sink" mechanism. This ensures that for tasks requiring perfect recall (e.g., looking up a variable definition from a distant module), the language runtime has a fallback mechanism superior to pure recurrence. However, to maintain the $O(1)$ inference guarantee, these attention layers would likely need to be windowed or restricted, or used only during "compilation" (context loading) rather than continuous generation.

---

# 3. Implementation Plan

This section outlines the engineering blueprint for **Cortex**. To achieve maximum performance and control, we adopt a **zero-dependency** philosophy. We will not rely on heavy frameworks like torch, candle, or burn. Instead, we will build a bare-metal tensor library and SSM kernel from scratch using the advanced features available in **Rust 1.93.0**.

## 3.1 The Tensor Foundation

The foundation of Cortex is CortexTensor, a minimalist tensor library tailored specifically for the operations required by SSMs (Strided views, Matrix Multiplication, Element-wise ops, and Scans).

### 1. Memory Layout and Strided Views

In Rust, we represent a tensor not as a deep object hierarchy but as a struct managing a contiguous block of memory. We use Vec (or f16 / bf16 via half crate or raw bytes) combined with a shape and strides vector.

```
// cortex_tensor/src/lib.rs
use std::alloc::{Layout, alloc, dealloc};

pub struct Tensor {
// Pointer to the raw data buffer (aligned for SIMD)
data: *mut T,
// Dimensions of the tensor
shape: Vec,
// Strides for navigating dimensions without copying
strides: Vec,
// Total number of elements
numel: usize,
// Device placement (CPU, GPU via FFI)
device: Device,
}

pub enum Device {
Cpu,
Cuda(usize), // Requires low-level FFI to CUDA driver API
Metal, // For Apple Silicon support via objc/metal-rs
}
```

**Optimization Insight:** SSMs heavily rely on "causal" masking and time-shifting. We implement **Strided Views** to allow reshaping, slicing, and time-shifting without copying data. For example, accessing the previous state $x_{k-1}$ should be a pointer arithmetic operation, not a memory copy. This is critical for the $O(1)$ inference promise.23

### 2. SIMD Integration (Rust 1.93.0)

Rust 1.93.0 provides stable and mature access to SIMD (Single Instruction, Multiple Data) operations, which are essential for performance on CPUs. We will leverage std::simd (Portable SIMD) where possible for cross-platform

compatibility, and fall back to std::arch for architecture-specific intrinsics (AVX-512 on x86, NEON on ARM).

- **The Scan Bottleneck:** The core "Scan" operation involves sequential dependencies ($x_k$ depends on $x_{k-1}$). A naive loop is too slow.
- **SIMD Strategy:** While we cannot parallelize *across* time in a simple scan, we can parallelize *across* the state dimension $D$. The state $x \in \mathbb{R}^N$ is a vector of size $L$ (e.g., 16 or 128). We can load multiple elements of $x \in \mathbb{R}^N$ into a single SIMD register (e.g., a 512-bit ZMM register can hold 16 f32 values).
- **Implementation:** We define a SimdBackend trait. On x86_64, we implement this using std::arch::x86_64::*; to leverage **AVX-512 Fused-Multiply-Add (FMA)** instructions. FMA is critical because the SSM update $x_{k+1} = \bar{A}_k x_k + \bar{B}_k u_k$ is fundamentally a multiplication followed by an addition. Doing this in a single cycle reduces latency and rounding errors.25

## 3.2 The SSM Kernel

This is the "heart" of Cortex. We must implement the kernel that performs the time-evolution of the system. We target the **Mamba-2 SSD algorithm** for its hardware efficiency, but understanding the parallel scan is a prerequisite.

### The Parallel Associative Scan (Blelloch Algorithm)

The recurrence relation $x_k = \bar{A}_k x_{k-1} + \bar{B}_k u_k$ appears strictly sequential. However, it can be parallelized during training (when all inputs are known) using the **Parallel Associative Scan** (also known as the Blelloch Scan).27

We define a binary operator $\bullet$ that acts on tuples of (Matrix, Vector). Let the tuple at time $k$ be $(\bar{A}_k,\ \bar{B}_k u_k)$. The operator is defined as:

$$(a_1, x_1) \bullet (a_2, x_2) = (a_2 \cdot a_1,\ a_2 \cdot x_1 + x_2)$$

This operator is **associative** ($ (a \bullet b) \bullet c = a \bullet (b \bullet c) $). This associativity allows us to compute the prefix sums (the sequence of states) using a tree-based reduction in $O(\log L)$ time, rather than linear $O(L)$ time.

**Rust Implementation Details:**

1. **Up-Sweep (Reduction):** We build a binary tree over the sequence chunks. Each node computes the aggregate transition matrix and effective input for its children.
2. **Down-Sweep:** We traverse the tree downwards, passing accumulated states to the leaves to compute the final state at every timestep.
3. **Chunking:** To optimize for CPU cache (L1/L2) and GPU Shared Memory (SRAM), we do not run the tree scan over individual tokens. We **chunk** the sequence (e.g., blocks of 1024 tokens). We process the chunks serially (fast in registers) and parallelize the scan *across* the chunks using Rust's rayon library for thread pooling.27

```
// cortex_kernel/src/scan.rs
// Simplified conceptual logic for the associative operator
// Inputs are tuples of (A, Bx) representing the linear recurrence parameters
fn combine_states(left: (f32, f32), right: (f32, f32)) -> (f32, f32) {
let (a_l, x_l) = left;
let (a_r, x_r) = right;
// The operator definition:
// New A = A_right * A_left
// New X = A_right * x_left + x_right
(a_r * a_l, a_r * x_l + x_r)
}
// This allows us to use Rayon to fold/reduce chunks in parallel
```

**Implementing Mamba-2 SSD**

For the highest performance, we implement the SSD algorithm. This treats the sequence mixing as a matrix multiplication.

- **Block Decomposition:** We view the lower-triangular mixing matrix $M$ as a collection of blocks.
- **Diagonal Blocks:** These represent intra-chunk interactions. Since $A$ is scalar-times-identity, these blocks are essentially small causal attention matrices. We compute these using standard matrix multiplication kernels.

- **Off-Diagonal Blocks:** These represent the carry-over from previous chunks. This is handled by a simplified "segment sum" scan.
- **Rust/CUDA FFI:** To perform the large matrix multiplications efficiently on GPU, Cortex will use the cust (CUDA support for Rust) crate or raw FFI to link against custom CUDA kernels written in inline PTX or C++ that utilize wmma (Warp Matrix Multiply Accumulate) instructions.19

## 3.3 The Mamba Block Architecture

We encapsulate the kernel in the Mamba Block structure, which forms the building block of the Cortex language model.

1. **Input Projection:** A Linear layer expanding the input dimension $d$ to $Ed$ (creating two branches: the signal branch and the gate branch).
2. **Convolution:** A 1D causal convolution (kernel size 4). This acts as a "local" look-back, ensuring the model sees a small window of context before the "global" SSM state processing.
    - *Zero-Dependency Challenge:* We must implement Conv1d manually. Since the kernel size is small (4), we implement this as a direct sliding window operation using SIMD shuffle instructions, avoiding the complexity of FFT-based convolution.13
3. **Discretization:** We compute the discrete parameters $\bar{A}_k$ and $\bar{B}_k$ from the input-dependent predictions. This requires efficient exp (exponential) and element-wise multiplication functions in our tensor library.
4. **SSM Execution:** We run the selective_scan kernel (or SSD matrix multiplication) on the discretized parameters.
5. **Gating:** The output of the SSM is multiplied element-wise by the gate branch (which has passed through a SiLU activation). This gating allows the model to control the flow of information deeper into the network.
6. **Output Projection:** A final Linear layer projects the data back to dimension $d$.13

## 3.4 The Language Interface

Cortex is defined as an "AI-first" language. This means the programmer defines **Types** and **Constraints**, and the SSM acts as the "solver" or "generator" at runtime.

### Grammar-Guided Generation (Constrained Decoding)

To make the output of the SSM executable and safe, Cortex integrates a **Context-Free Grammar (CFG)** engine directly into the inference loop.31 This is the "Oracle" type system.

1. **The "Oracle" Type System:**
   In Cortex, a function definition acts as a prompt, and the type signature acts as a hard constraint.
   Rust
   // Cortex Language Pseudo-code
   // The model must generate a valid u8 integer between 0 and 120.
   fn get_user_age(bio: String) -> u8 constrained_by Range(0, 120);
2. **The Masking Engine:**
   At every generation step $k$, the Cortex runtime checks the current partial output against the defined grammar (or Regex/Type). It computes a **logit mask**.
   - If the parser expects a digit to satisfy the u8 type, all tokens in the vocabulary that are *not* digits are masked (their logit is set to $-\infty$).
   - This guarantees that the output *always* matches the type signature. The model effectively "hallucinates" a valid program trace that satisfies the type constraints.33

**Implementation:** We implement a parser combinator in Rust (or adapt nom or pest concepts) that maintains parsing state incrementally. As the SSM generates tokens, the parser advances. If a token would cause a parse error, it is forbidden *before* the sampling step. This transforms the probabilistic LLM into a deterministic, type-safe execution engine.

---

# 4. Hardware-Aware Implementation

The theoretical elegance of SSMs—infinite context, constant inference—falls apart if not mapped correctly to the physical reality of modern hardware. The bottleneck in modern AI is **Memory Bandwidth**, not Compute (FLOPS).

## 4.1 The Memory Hierarchy: HBM vs. SRAM

Modern GPUs (like the H100) have massive High Bandwidth Memory (HBM) (e.g., 80GB) but extremely limited, ultra-fast Static RAM (SRAM) (e.g., ~200KB per Streaming Multiprocessor).

- **Standard Transformer Failure Mode:** Transformers operate by loading the massive Attention Matrix ($QK^T$) from HBM to compute attention scores. This saturates the memory bandwidth, leaving the compute cores idle.
- **Mamba's Hardware-Awareness:** The Cortex kernel must implement **Kernel Fusion**. Instead of writing intermediate results (like the discretized matrices $(\bar{A}_k, \bar{B}_k)$) back to HBM, we load the SSM parameters ( $(A, B, C, \Delta)$) from HBM into SRAM once. We then perform the entire recurrence (scan) logic *inside* the fast SRAM registers. We only write the final result back to HBM. This reduces Memory I/O by orders of magnitude, making the operation compute-bound rather than memory-bound.11

## 4.2 Tensor Cores and Block Decomposition

Rust 1.93.0 allows us to interface with the GPU at a low level. To utilize NVIDIA Tensor Cores (which specialize in $16 \times 16$ half-precision matrix multiplies), we cannot simply run the Blelloch scan, as it is a scalar operation.

- **SSD Adoption:** This is the primary reason for adopting the Mamba-2 SSD algorithm.
- **Decomposition:** We decompose the long sequence into chunks (blocks) of size $Q$ (e.g., 128 or 256).
- **Matrix Form:** Within a block, the state update $Y_{\text{block}} = M_{\text{block}} \cdot X_{\text{block}}$ looks exactly like a matrix multiplication.

- **Dispatch:** We dispatch these block-level operations to Tensor Cores using wmma intrinsics.
- **Connection:** We connect the independent blocks using the scalar scan logic (which is cheap compared to the heavy lifting). This hybrid approach allows Cortex to utilize the specialized silicon designed for Transformers to accelerate SSMs.19

### 4.3 CPU Inference with AVX-512

While GPUs are preferred for training, Cortex aims to be a general-purpose language, meaning it must run efficiently on CPUs.

- **Vertical Fusion:** On the CPU, we fuse the input projection, convolution, and SSM scan into a single loop over the sequence length. This ensures that once a token is loaded into the L1 cache, all operations associated with it are performed before it is evicted.
- **SIMD Lanes:** We map the state dimension $D$ (e.g., 64) to SIMD lanes. A single AVX-512 register holds 16 floats. We process the state updates for 16 channels simultaneously. This provides a theoretical 16x speedup over scalar code, crucial for achieving acceptable latency on standard server processors.25

## 5. Conclusion

Cortex represents a fundamental divergence from the Transformer-dominated era of AI. By returning to the theoretical roots of Control Theory—specifically the State Space Models established in the 1960s—and modernizing them with **State Space Duality** and **Parallel Associative Scans**, we can build an AI runtime that is linear in complexity, theoretically infinite in context, and strictly typed in output.

The implementation plan provided leverages the stability and low-level control of **Rust 1.93.0** to eliminate the bloat of Python-based frameworks. By manually managing memory hierarchies (HBM vs. SRAM) and implementing custom kernels for the SSD algorithm, Cortex results in a system where the AI is not a

"black box" called via an external API, but a deterministic, compile-time verifiable component of the software stack. This fusion of rigorous systems programming with probabilistic state modeling defines the new category of the **AI-first programming language**.

## References

[1] What Are State Space Models? | IBM. https://www.ibm.com/think/topics/state-space-model [2] Along comes a Mamba: an evolution in sequence models based on state space models | by Wilbur de Souza | Medium. https://medium.com/@wilburdes/along-comes-a-mamba-an-evolution-in-sequence-models-based-on-state-space-models-2bd3d0e02d86 [3] State-Space Models: Learning the Kalman Filter - Andrew Fairless, Ph.D. https://afairless.github.io/page/learning-kalman-filter/ [4] State Space Models (1): introduction to SSMs - Chus Antonanzas. https://chus.space/blog/2024/ssm_1_context/ [5] Comprehensive Breakdown of Selective Structured State Space Model — Mamba (S5). | by Freedom Preetham | Autonomous Agents | Medium. https://medium.com/autonomous-agents/comprehensive-breakdown-of-selective-structured-state-space-model-mamba-s5-441e8b94ecaf [6] Discretization of State-Space System Models - University of Washington. https://faculty.washington.edu/chx/teaching/me547/1-8_zohSS_slides.pdf [7] HiPPO: Recurrent Memory with Optimal Polynomial Projections | NSF Public Access Repository. https://par.nsf.gov/biblio/10214620-hippo-recurrent-memory-optimal-polynomial-projections [8] Gu, A., et al. (2020). "HiPPO: Recurrent Memory with Optimal Polynomial Projections." *arXiv:2008.07669*. https://arxiv.org/abs/2008.07669 [9] HiPPO: Recurrent Memory with Optimal Polynomial Projections - Hazy Research. https://hazyresearch.stanford.edu/hippo [10] HiPPO: Recurrent Memory with Optimal Polynomial Projections - arXiv. https://arxiv.org/pdf/2008.07669 [11] What Is A Mamba Model? | IBM. https://www.ibm.com/think/topics/mamba-model [12] A Visual Guide to Mamba and State Space Models - Maarten Grootendorst. https://www.maartengrootendorst.com/blog/mamba/ [13] Here Comes Mamba: The Selective State Space Model | Towards Data Science.

https://towardsdatascience.com/here-comes-mamba-the-selective-state-space-model-435e5d17a451/ [14] Mamba architecture : A Leap Forward in Sequence Modeling | by Puneet Hegde - Medium. https://medium.com/@puneetthegde22/mamba-architecture-a-leap-forward-in-sequence-modeling-370dfcbfe44a [15] Mamba: Linear-Time Sequence Modeling with Selective State Spaces - OpenReview. https://openreview.net/forum?id=tEYskw1VY2 [16] How Mamba Beats Transformers at Long Sequences - Galileo AI. https://galileo.ai/blog/mamba-linear-scaling-transformers [17] Is Mamba inference faster than Transformers? (in practice) : r/LocalLLaMA - Reddit. https://www.reddit.com/r/LocalLLaMA/comments/1fli4t3/is_mamba_inference_faster_than_transformers_in/ [18] State Space Duality (Mamba-2) Part III - The Algorithm | Tri Dao. https://tridao.me/blog/2024/mamba2-part3-algorithm/ [19] Mamba2: The Hardware-Algorithm Co-Design That Unified Attention and State Space Models | by Daniel Stallworth | Medium. https://medium.com/@danieljsmit/mamba2-the-hardware-algorithm-co-design-that-unified-attention-and-state-space-models-77856d2ac4f4 [20] Dao, T., & Gu, A. (2024). “Transformers are SSMs: Generalized Models and Efficient Algorithms Through Structured State Space Duality.” *arXiv:2405.21060*. https://arxiv.org/abs/2405.21060 [21] Jamba: A Hybrid Transformer-Mamba Language Model - arXiv. https://arxiv.org/html/2403.19887v1 [22] Architectural Evolution in Large Language Models: A Deep Dive into Jamba’s Hybrid Transformer-Mamba Design - Greg Robison. https://gregrobison.medium.com/architectural-evolution-in-large-language-models-a-deep-dive-into-jambas-hybrid-transformer-mamba-c3efa8ca8cae [23] ramsyana/RustTensor: A learning-focused, high-performance tensor computation library built from scratch in Rust, featuring automatic differentiation and CPU/CUDA backends. - GitHub. https://github.com/ramsyana/RustTensor [24] reinterpretcat/zero-depend-pub: An educational Rust workspace featuring zero-dependency crates built using only standard library - GitHub. https://github.com/reinterpretcat/zero-depend-pub [25] Parallel Prefix Sum with SIMD - arXiv. https://arxiv.org/html/2312.14874v1 [26] The state of SIMD in Rust in 2025 | by Sergey “Shnatsel” Davidoff | Medium. https://shnatsel.medium.com/the-

state-of-simd-in-rust-in-2025-32c263e5f53d [27] Mamba No. 5 (A Little Bit Of...) - Sparse Notes. https://jameschen.io/jekyll/update/2024/02/12/mamba.html [28] Parallel scans | ex.rs. https://ex.rs/parallel-scans/ [29] Mamba: Linear-Time Sequence Modeling with Selective State Spaces - arXiv. https://arxiv.org/pdf/2312.00752 [30] Mamba-2: Algorithms and Systems | Princeton Language and Intelligence. https://pli.princeton.edu/blog/2024/mamba-2-algorithms-and-systems [31] CRANE: Reasoning with constrained LLM generation - OpenReview. https://openreview.net/forum?id=wKs9fHYxCV [32] Improving LLM Code Generation with Grammar Augmentation - arXiv. https://arxiv.org/html/2403.01632v1 [33] structured decoding, a guide for the impatient. https://aarnphm.xyz/posts/structured-decoding [34] Achieving Efficient, Flexible, and Portable Structured Generation with XGrammar - MLC. https://blog.mlc.ai/2024/11/22/achieving-efficient-flexible-portable-structured-generation-with-xgrammar [35] Mamba: The Linear-Time Architecture That Could Challenge Transformers - Medium. https://medium.com/@danieljsmit/mamba-the-linear-time-architecture-that-could-challenge-transformers-b27cf1b0dac1 [36] parallel prefix (cumulative) sum with SSE - Stack Overflow. https://stackoverflow.com/questions/19494114/parallel-prefix-cumulative-sum-with-sse

[37] Charlot, D.J. “Bounded Entropy in Formal Languages: A Mathematical Foundation for Deterministic Code Generation.” OpenIE Technical Report, December 2025.

[38] Charlot, D.J. “Deterministic Code Auditing for Production-Ready Software.” OpenIE Technical Report, December 2025.

[39] Charlot, D.J. “Cortex: Neural-Symbolic Programming for Energy-Efficient Code Execution.” OpenIE Technical Report, January 2026.

[40] Charlot, D.J. “Metabolic Cascade Inference: Hardware-Aware Adaptive Routing for Energy-Efficient AI.” OpenIE Technical Report, January 2026.

[41] Charlot, D.J. “The AI Inference Crisis: Why Current Approaches Are Unsustainable.” OpenIE Technical Report, January 2026.

---

*This whitepaper describes independent academic research focused on AI-first programming language design using State Space Models. Published freely without patent protection under CC BY-ND 4.0 license.*

**Contact**: david@openie.dev | dcharlot@ucsb.edu **Latest Updates**: https://openie.dev/projects/cortex