Efficient Implementation of Mamba Architecture for Radio-based Localisation Using LuViRA datasest

Chenxi Zhang and Shengjie Chen

Blue text in the Abstract, Introduction, Conclusion, and Section II is by Chenxi; all remaining sections are by Shengjie.

***Abstract*—This paper investigates three Mamba-based models variant on the LuViRA [1] dataset and demonstrates the effectiveness of SSM-based architectures for indoor radio localization. Building on these findings, we propose task-oriented simplifications that preserve comparable inference accuracy while significantly reducing model size, hardware cost, and end-to-end training/inference overhead. To further enable deployment, we implement a synthesizable FPGA accelerator for the state-space model (SSM) hotspot in Mamba, targeting indoor radio localization (LuViRA). Software profiling identifies the SSM forward path as a dominant runtime contributor. We implement SSM as a streaming pipeline using three primitives (MAC, EWA, EWM) and a LUT-based sigmoid, with pipelined MVM arrays, interleaving memory access, and FIFO-based AXI-Stream control for backpressure-safe composition. The integrated module closes timing at 400 MHz using 76 DSPs and 6,244 LUTs, with an estimated on-chip power of 1.44 W, providing a baseline toward end-to-end Mamba inference. To support reproducibility, the accelerator implementation is open-sourced at https://github.com/Elison-debug/slim_mamba.**

***Keywords*—Mamba, state-space model (SSM), indoor localization, massive MIMO, FPGA acceleration.**

I. Introduction

INDOOR radio localization requires modeling high-dimensional wireless signals with strong temporal dynamics and multipath effects. As antenna/channel dimensions increase, conventional FCNN-based regressors become inefficient: their parameter count and compute cost grow rapidly with input width, making deployment and scaling difficult [2]. This motivates architectures that can process long signal sequences with stronger parameter efficiency.

Mamba, as a modern state-space-model (SSM) architecture, offers a compact alternative to dense FCNNs by summarizing temporal context through recurrent state updates with linear-time sequence processing. For indoor localization, this enables efficient use of temporal dynamics without the parameter explosion of FCNNs.

In this project, we follow a software-to-hardware co-design path on the LuViRA dataset [1] and evaluate three Mamba-family variants for radio localization: vanilla Mamba v1 [3], channel mamba [4], and slim mamba. We first adopted a channel mamba design [4] to support multi-domain feature fusion (e.g., CIR real/imag, power, and optional differential features), motivated by potential cross-channel interaction and extensibility to future multi-modal inputs (e.g., audio). However, experiments show that, under the current setting, additional channel-fusion branches bring only limited accuracy gain while introducing substantial overhead in model complexity, training time, and hardware mapping cost. For clarity, we refer to the channel-fusion variant as *channel mamba* and the simplified model as *slim mamba* throughout the report. Overall, slim mamba achieves comparable test accuracy to channel mamba and the FCNN baseline while reducing the model size to 3.40 MB (0.77% of FCNN) and the training time to 13 s per epoch (Table II).

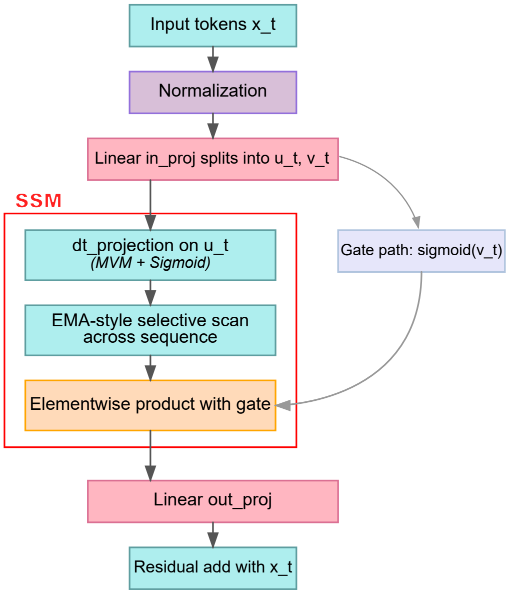

Fig. 1. Algorithmic view of the slim mamba block [3].

Unlike FCNN baselines that perform single-snapshot localization, Mamba processes a short sequence of observations jointly, enabling multi-snapshot localization and exploiting temporal dynamics in the wireless channel. Based on this observation, we first examined the temporal context length within the channel mamba prototype and found that a short window ($K = 16$) already achieves strong accuracy. This indicates that very long-range memory is not necessary for this short-horizon localization task and may introduce redundancy. We therefore proposed a hardware-friendly simplification of Mamba v1 that preserves comparable inference accuracy while reducing model size and end-to-end runtime overhead. The final slim mamba architecture is in Fig. 1.

From a deployment perspective, acceleration should target real runtime hotspots. Profiling (Table I) shows that the SSM forward path dominates execution, so we implement it with a fully synthesizable FPGA design. The accelerator uses a streaming pipeline built from MAC, EWA, and EWM operators plus a LUT-based sigmoid, and scales via a $4 \times 4 \times 4$ pipelined MVM array, interleaving memory access, and FIFO-

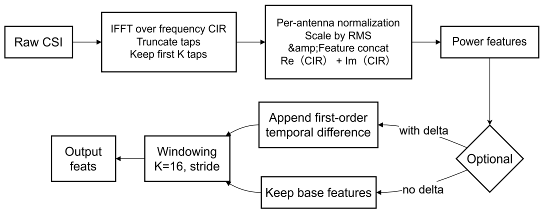

Fig. 2. Data preprocessing pipeline from CSI to model-ready features.

based AXI-Stream flow control for backpressure-safe composition.

Although recent studies have explored Mamba acceleration from complementary angles, including hardware–algorithm co-design [5], reconfigurable operator support [6], and hybrid dataflows [7], several trade-offs remain under-discussed for radio-localization-oriented FPGA deployment. First, compute-array organization is often fixed without controlled comparison under identical workload assumptions. Second, memory organization for mixed linear and element-wise SSM pipelines is not sufficiently analyzed, especially for interleaved banked access. Third, control granularity trade-offs between coarse scheduling and fine-grained streaming are rarely quantified under the same constraints.

Motivated by these gaps, this work contributes: 1) an empirical software-side study from channel mamba to slim mamba for short-window localization, 2) a hotspot-driven FPGA mapping of the SSM datapath, and 3) a consistent evaluation of array, memory, and control trade-offs. The rest of this report first presents SSM decomposition and hardware mapping, then details the accelerator architecture and implementation results.

TABLE I
FUNCTION-LEVEL PROFILING SUMMARY.

Function	Time	Share (%)
Linear projection (in/out)	30	7.75
Normalization (RMSNorm)	57	14.73
SSM dt projection (MVM + sigmoid)	140	36.18
SSM selective scan (state update)	160	41.34
Total (listed)	387	100.00

II. SOFTWARE EXPLORATION AND TASK-DRIVEN SIMPLIFICATION

This section presents the software-side exploration that guided our final model choice and hardware-oriented redesign. We start from a channel mamba architecture for multi-domain feature fusion, then progressively simplify the model based on empirical evidence under the LuViRA workload.

A. Task Setting and Data Pipeline

We target indoor localization on LuViRA, where each recording is a time–frequency–space CSI tensor ($T \approx 4000$ at 100 Hz, 100 antennas, 100 subcarriers). To align software processing with deployment-oriented constraints, we organize the dataflow in a hardware-friendly manner with sequential window access and compact per-frame storage.

Raw CSI is converted into model-ready features through: (1) IFFT-based CIR extraction over subcarriers, (2) tap truncation, and (3) feature construction using CIR real/imaginary components, per-antenna log-power, and optional first-order temporal difference(see Fig. 2). For each trajectory/grid file, we store `base.feats.npy`, `base.xy.npy`, and `base.ts.npy`, while `base.json` records role-aware window indices (train/eval/test), window length, stride, and feature configuration. This layout supports on-demand window generation without offline materialization of all training samples.

In our final experiments, we enable `--preload`, which significantly improves training throughput by avoiding repeated data fetch/transfer overhead across iterations. At the same time, the pipeline still supports lazy-loading mode for resource-constrained platforms, and we verified stable operation on a low-end MX330 GPU setup. Although preload increases memory usage, the overhead remains acceptable for our workload (GPU memory usage rises from approximately 0.3 GB to 1.4 GB), while the training speedup is substantial. Finally, we compute train-only global normalization statistics (`stats_train.npz`, mean/std with std-floor) to ensure stable scaling across grids and avoid evaluation/test leakage.

B. Initial Design: Channel mamba for Multi-domain Fusion

Our initial hypothesis was that channel mamba could better exploit heterogeneous input domains (e.g., CIR, power, and differential signals), while also providing a natural extension path to future multi-modal inputs such as audio. Accordingly, we first implemented a multi-branch channel mamba model with explicit channel fusion blocks. On LuViRA, this model achieved strong localization accuracy (test mean err = 12.9 cm) with a compact serialized size of 40.4 MB, which is substantially smaller than the 440 MB FCNN model [8] reported in prior work (Table II).

C. Empirical Reassessment

Despite the above motivation, the extra fusion complexity yields only marginal gains while increasing parameter count, training cost, and hardware mapping overhead. In addition, differential features bring limited improvement relative to their added structural complexity, which weakens their deployment value. From a deployment perspective, channel mamba also poses practical challenges on FPGA: it relies on several complex nonlinear operators that complicate quantization, stacks multiple channel mamba blocks that amplify hardware cost, and inherits GPU-oriented kernels that are not optimized for FPGA dataflow. Moreover, the original Mamba baseline shows a pronounced generalization gap (training improves while test accuracy lags), as reflected by the large eval/test discrepancy in Table II. Vanilla Mamba v1 also performs markedly worse than channel mamba on this dataset (Table II), suggesting that its long-context design is not a good fit for short-horizon localization. Within the channel mamba baseline, a short context ($K = 16$) already achieves strong performance, so we use it as a practical upper bound for temporal context before simplifying the model.

Qualitatively, Fig. 3 visualizes the spatial error distribution for a representative grid, highlighting regions with larger localization bias.

TABLE II
COMPARISON OF MODEL VARIANTS ON LUVIRA (MEASURED RESULTS).

Model	Input Features	Model Size (MB)	Train Time / epoch (s)	it/s	Test mean err (cm)	HW Cost Proxy
FCNN (prior work) [8]	CIR	440	N/A	N/A	~ 13	High
channel mamba	CIR + Power	40.4	73	13.18	12.9	High
Vanilla Mamba v1	CIR + Power	3.38	33	32.62	24.74	Medium
slim mamba	CIR + Power	3.40	13	75.30	13.5	Low

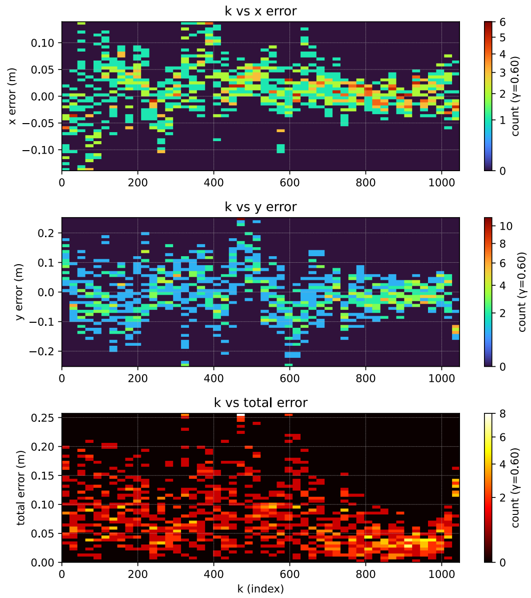

Fig. 3. Error heatmap for grid 102.

D. Task-driven Simplification of Mamba

We then revisited vanilla Mamba and observed a clear overfitting pattern: the training loss continues to decrease while generalization remains poor. The quantitative gap between training and test performance is summarized in Table II. This motivates a task-driven simplification that reduces model capacity and regularizes the temporal computation while keeping the essential selective recurrence. Based on these findings, we simplify Mamba v1 into a hardware-friendly form by removing non-essential complexity while preserving the core recurrent update behavior required by the task.

E. Final Model and Key Differences

Compared with channel mamba, the final model keeps the core recurrent update path but adopts a more regular and deployment-friendly structure (fewer branches, reduced state/update complexity, and more hardware-compatible operators). This redesign preserves localization accuracy comparable to channel mamba and the FCNN baseline reported in Table II, while substantially reducing training and inference time, hardware cost, and quantization difficulty. The slim mamba SSM follows an EMA-style input-conditioned recurrence, followed by a SiLU gate and a 1x1 output projection. This removes the full (A, B, C, D) parameterization and low-rank discretization while retaining the temporal update structure (see Fig. 1). The error distribution is summarized as a CDF in Fig. 4, showing the proportion of samples below a given error threshold.

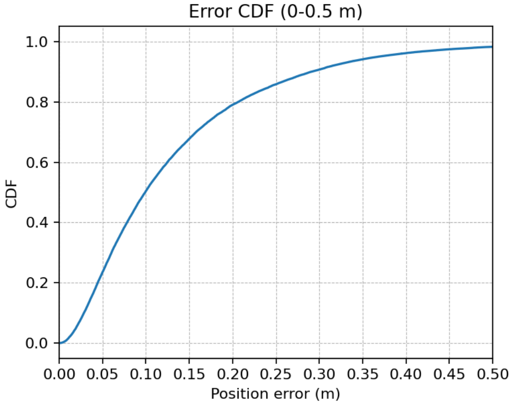

Fig. 4. CDF of localization error.

F. Window Length and Patch-Length Ablation

After fixing slim mamba, we ablated the temporal window length K together with patch length/stride. Table IV shows that $K = 1$ and $K = 2$ achieve the best test accuracy (0.135 m). We attribute this to the dense sampling in LuViRA: short-term dynamics carry strong cues, whereas longer windows add redundant context and can even hurt generalization in slim mamba. We therefore adopt $K \leq 2$ as the default setting for slim mamba in the remainder of the software study.

III. SSM DECOMPOSITION AND HARDWARE MAPPING

From an algorithmic perspective, the SSM core can be decomposed into three major steps: (1) *dt_projection*, which derives step-size coefficients Δ_t from the current input; (2) *state update*, the time-recursive selective scan (recurrent update) that fuses the current input with the previous state to produce a new state s_t; and (3) *output gate*, which combines the updated state with the gating branch to form the output. This decomposition is consistent with Fig. 1, where the input is split into a main branch and a gate branch after the in_proj layer; the main branch is locally modulated and then fed into the SSM for state update, while the output is produced via element-wise interaction with the gate.

When mapped to hardware, the SSM can be reduced to three key hardware primitives:

- **MAC** (matrix–vector multiply–accumulate) for dt_proj.

$$\lambda_t = \sigma(W_{dt}\Delta u_t + b) \tag{1}$$

TABLE III
MODULE-LEVEL DIFFERENCES BETWEEN CHANNEL MAMBA AND SLIM MAMBA.

Component	**Channel mamba**	**Slim mamba (Ours)**	**Impact**
Input fusion path	Multi-branch, channel-fusion-oriented backbone with extra structural overhead.	Single-path backbone with no explicit channel-fusion branches.	Lower control complexity and easier hardware scheduling.
State update (SSM)	Full selective scan parameterization with higher arithmetic and control complexity.	Hardware-friendly selective recurrence with reduced state-update complexity.	Faster execution and easier FPGA mapping.
Temporal context usage	Same short-context setting but more complex fusion paths.	Same short-context setting with simplified recurrence.	Better task fit; less redundancy and lower overfitting risk.
Operator pattern	Mixed operators across fusion branches with heavier scheduling burden.	More regular Conv/element-wise-friendly computation flow.	Better pipelining and implementation regularity.
Model size / compute	Higher parameter and compute cost.	Lower parameter and compute cost.	Reduced memory footprint and MAC demand.
Quantization difficulty	Multiple nonlinear paths and branch interactions complicate calibration.	Fewer branches with clearer activation flows.	Easier calibration and hardware quantization.

TABLE IV
ABLATION ON WINDOW LENGTH K WITH PATCH LENGTH/STRIDE VARIANTS (LUVIRA).

K (pl, s)	**Eval mean err (m)**	**Test mean err (m)**
1 (1,1)	0.10068	**0.135**
2 (2,2)	0.09557	**0.135**
4 (2,2)	0.09814	0.139
8 (2,2)	0.09405	0.141
4 (4,4)	0.09983	0.137
6 (4,4)	0.09839	0.138
8 (4,4)	0.09749	0.144
8 (8,4)	0.09744	0.152
12 (8,4)	0.09286	0.156
16 (8,4)	0.09225	0.161
24 (8,4)	0.09432	0.159
32 (8,4)	0.09328	0.166

- **EWA** (element-wise add) for state update.

$$s_t = \lambda_t \odot s_{t-1} + (1 - \lambda_t) \odot u_t \tag{2}$$

- **EWM** (element-wise multiply) for output gating.

$$y_t = s_t \odot g_t \tag{3}$$

Therefore, our design builds a streaming accelerator using **MVM (MAC) + EWA + EWM** as the core building blocks, centered on the SSM state-update hotspot.

Fig. 5 illustrates the top-level data path of our accelerator, which can be summarized into four stages. **(i) Linear compute and reduction**: the input vector is streamed from `xt_buf`, weights are supplied by `WBUF`, and a 4×4×4 MAC array produces partial sums that are reduced by a reduction tree, forming the linear-layer outputs. **(ii) Nonlinearity and gate preparation**: the linear outputs are bias-adjusted, decoupled by a FIFO, and passed through a Sigmoid LUT to generate gate-related coefficients, aligning with the gate path. **(iii) State update (Element-wise update + s_buffer)**: a `Join` module aligns the required streams (e.g., λ and x_t) and feeds the element-wise update unit that performs combined **EWM/EWA** operations to produce s_{new}, with `s_buffer` closing the read-old/write-new loop for recurrent state updates. **(iv) Output gate**: the updated state and the gate stream are aligned by another `Join`, decoupled by a FIFO, and multiplied element-wise in the EWM gate to produce y_{out}.

This organization keeps the SSM workload concentrated on the **MAC array (MVM)** and **element-wise (EWA/EWM)** modules, while FIFO and AXI-streaming boundaries enable fine-grained pipelining and robust backpressure propagation.

IV. METHOD

A. MAC Array Design and Architecture Selection

This chapter presents the hardware implementation methods of our SSM accelerator, starting from the compute backbone: the MAC array. For the SSM datapath, linear layers dominate the arithmetic cost and can be reduced to an MVM of size $(256 \times 256) \cdot (256 \times 1)$, i.e., $256 \times 256 = 65{,}536$ MAC operations. A useful throughput bound is

$$CC_{\min} = \left\lceil \frac{\text{Total MAC operations}}{\#\text{PEs}} \right\rceil, \tag{4}$$

which makes explicit the fundamental trade-off between parallelism (DSP/PE count) and latency in cycles.

We compare three candidate array organizations.

1) A 16×16 systolic array offers strong spatial reuse for GEMM (both operands reuse across the 2D array) and can theoretically complete a tile in 256 cycles. For GEMV, weights have little spatial reuse (each $W_{i,k}$ is typically consumed once per input), making the design weight-bandwidth dominated: full utilization would require streaming ∼256 weights/cycle, which also increases routing pressure and complicates timing closure in practice.
2) A 64×1 single-column GEMV engine directly addresses the bandwidth bottleneck and is well matched to GEMV-style computation, achieving a 1024-cycle latency under a 64 weights/cycle budget. However, a full Mamba pipeline also includes matrix–matrix operations beyond the SSM core (e.g., input and output linear projection), so a one-column GEMV structure is less reusable for those kernels and is therefore suboptimal for long-term extensibility.
3) Our proposed 4×4×4 pipelined MAC array is a compromise: it preserves the 64 weights/cycle bandwidth

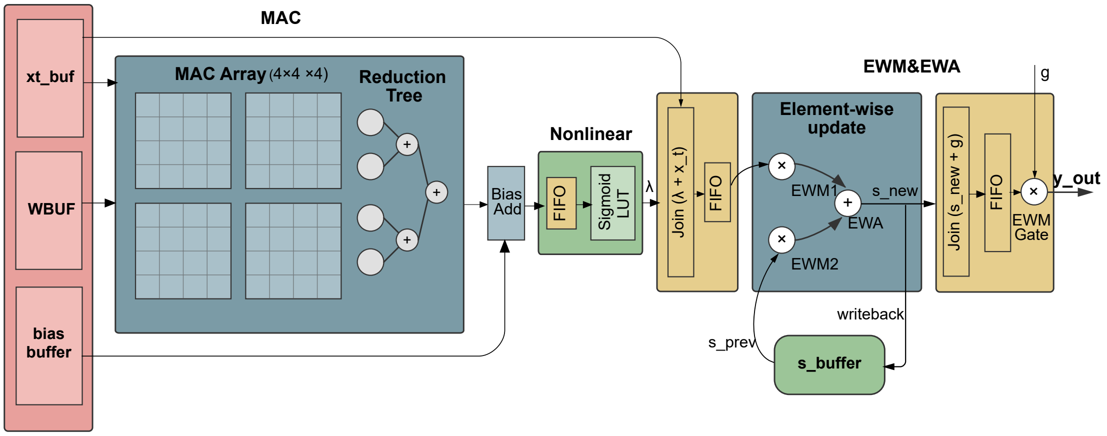

Fig. 5. Top-level architecture of the proposed accelerator (based on the Mamba/SSM operator structure) [3].

TABLE V
WEIGHT INPUT COLUMN SCHEDULING FOR THE 4×4×4 PIPELINED MAC ARRAY.

Cycle	Array1 Columns	Array2 Columns	Array3 Columns	Array4 Columns	Description
1	col0–3	–	–	–	ARRAY1 preloads first 4×4 block (tile1).
2	col16–19	col4–7	–	–	ARRAY2 begins tile1.
3	col32–35	col20–23	col8–11	–	ARRAY3 begins tile1 (3-cycle stagger).
4	col48–51	col36–39	col24–27	col12–15	ARRAY4 joins; pipeline full.
5–16	continue +16 stride	same	same	same	Steady-state loading of tile1.
17	**col256–259 → tile2**	col244–247	col232–235	col220–223	**ARRAY1 starts tile2 (row4–7).**
18	col272–275	**col260–263 → tile2**	col248–251	col236–239	**ARRAY2 switches to tile2.**
19	col288–291	col276–279	**col264–267 → tile2**	col252–255	**ARRAY3 switches to tile2.**
20	col304–307	col292–295	col280–283	**col268–271 → tile2**	**ARRAY4 switches; tile1 finishes.**
21–37	continue +16 stride	same	same	same	Steady-state operation for tile2.
38	**tile3 preload**	–	–	–	ARRAY1 starts tile3.
39	–	**tile3 preload**	–	–	ARRAY2 starts tile3.
40	–	–	**tile3 preload**	–	ARRAY3 starts tile3.
41	–	–	–	**tile3 preload**	ARRAY4 starts tile3 (3-cycle stagger).
42–58	continue +16 stride	same	same	same	Steady tile3 operation.

target and achieves a similar 1024-cycle-class latency as the 64×1 baseline, while improving routing scalability relative to a monolithic systolic array and enabling higher reuse across multiple kernels.

The compute engine instantiates four parallel 4×4 MAC sub-arrays (ARRAY1–ARRAY4), forming a 4×4×4 pipeline organization. Each 4×4 sub-array consumes a 4-element slice of the input vector x_t together with a 4×4 weight block per cycle. Each array performs a local 4×4 MVM and accumulates the partial sums from the last cycle. After 4 cycles, all four sub-arrays have be used and the pipeline is fully filled. Importantly, the sub-array does **not** directly output four independent column-wise partial sums. Instead, it maintains accumulation across the tile and produces an **already-accumulated** 4×4 **partial-result matrix** at its output interface. This 4×4 partial-result matrix is then forwarded to a dedicated **three-level reduction tree**, which hierarchically reduces the 4×4 block into the required vector-form output used by subsequent stages.

With this organization, the MAC fabric focuses on dense local accumulation, while the reduction tree provides a structured and timing-friendly path to obtain the final MVM vector output. After a short fill phase, all four sub-arrays run concurrently in steady state, hiding control overhead while keeping the compute fabric small (64 MACs total), which is favorable for FPGA timing closure. Table V provides the full weight-column schedule;

TABLE VI
COLUMN START PROGRESSION PER SUB-ARRAY.

Array	Column Start Points	Δ
Array1	0 → 16 → 32 → 48 → 64	+16
Array2	4 → 20 → 36 → 52	+16
Array3	8 → 24 → 40	+16
Array4	12 → 28	+16

TABLE VII
CONSTANT 12-COLUMN SPACING BETWEEN ADJACENT ARRAYS WITHIN THE SAME CYCLE.

Cycle	A1→A2 Δ	A2→A3 Δ	A3→A4 Δ
2	16–4 = **12**	–	–
3	32–20 = **12**	20–8 = **12**	–
4	48–36 = **12**	36–24 = **12**	24–12 = **12**
5	64–52 = **12**	52–40 = **12**	40–28 = **12**

Weights are scheduled in 4-column blocks. For each sub-array, the column start index advances by a fixed stride of +16 every cycle. Within the same cycle, the four sub-arrays access distinct column blocks with fixed offsets $\{0, 4, 8, 12\}$, resulting in a constant 12-column spacing between adjacent

TABLE VIII
x_t INPUT SCHEDULING ALIGNED WITH THE WAVEFRONT TILE SWITCH ACROSS SUB-ARRAYS.

Cycle	Array1	Array2	Array3	Array4	Description
1	xt[0:3]	–	–	–	ARRAY1 begins tile1 (xt block0).
2	xt[0:3]	xt[0:3]	–	–	ARRAY2 joins tile1.
3	xt[0:3]	xt[0:3]	xt[0:3]	–	ARRAY3 joins tile1.
4	xt[0:3]	xt[0:3]	xt[0:3]	xt[0:3]	ARRAY4 joins tile1; steady begins.
5–16	xt[0:3]	xt[0:3]	xt[0:3]	xt[0:3]	tile1 steady-state.
17	**xt[4:7]**	xt[0:3]	xt[0:3]	xt[0:3]	**ARRAY1 starts tile2 (next row-block).**
18	xt[4:7]	**xt[4:7]**	xt[0:3]	xt[0:3]	**ARRAY2 switches to tile2.**
19	xt[4:7]	xt[4:7]	**xt[4:7]**	xt[0:3]	**ARRAY3 switches to tile2.**
20	xt[4:7]	xt[4:7]	xt[4:7]	**xt[4:7]**	**ARRAY4 switches; tile1 finishes.**
21–33	xt[4:7]	xt[4:7]	xt[4:7]	xt[4:7]	tile2 steady-state.
34	**xt[8:11]**	xt[4:7]	xt[4:7]	xt[4:7]	ARRAY1 starts tile3.
35	xt[8:11]	**xt[8:11]**	xt[4:7]	xt[4:7]	ARRAY2 switches.
36	xt[8:11]	xt[8:11]	**xt[8:11]**	xt[4:7]	ARRAY3 switches.
37	xt[8:11]	xt[8:11]	xt[8:11]	**xt[8:11]**	ARRAY4 switches; tile2 ends.

arrays. Table VI summarizes the per-array progression; and Table VII validates the constant 12-column spacing.

x_t input scheduling and wavefront tile switch: The input vector x_t is streamed from `xt_buf` in 4-element blocks. During pipeline fill (cycles 1–4), ARRAY1 starts first and ARRAY2/3/4 join sequentially; in steady state, all arrays consume the same x_t block for the active tile. When moving to the next tile, the x_t block update is aligned with the weight schedule and propagates as a wavefront across the arrays, minimizing global bubbles. Table VIII details the x_t schedule and matches the tile-switch cycles in Table V.

Overall, the 4×4×4 pipelined array provides a favorable balance among DSP usage, bandwidth demand, latency, and reuse. However, the compromise shifts complexity to the memory subsystem. The next subsection therefore focuses on the memory-access design and compares single-port versus dual-port organizations in terms of power and resource cost.

B. Interleaved Memory Bank Design

Sustaining the 4×4×4 pipeline MVM array in steady state requires the weight buffer (WBUF) to deliver stall-free parallel bandwidth, where four sub-arrays fetch one 4×4 column block per cycle. This section formalizes two conflict types and derives single-port and dual-port realizations under a unified modulo-unrolling mapping [9].

Conflict model: Space-conflict captures same-cycle port contention when multiple arrays access the same bank. A single-port bank supports at most one read per cycle, hence all four arrays must hit distinct banks per cycle. A dual-port bank supports up to two reads per cycle, allowing two arrays to share one bank.

Temporal-conflict captures cross-cycle reuse hazards under a finite read latency L. A sufficient safety condition is

$$\text{bank reuse distance} \geq L, \tag{5}$$

which prevents issuing a new access to a bank before prior reads have been safely resolved in the pipeline.

Modulo-unrolling mapping derived from the array dataflow: Weights are accessed in 4-column blocks, each corresponding to a 4×4 weight block. Define

$$block\_id = \left\lfloor \frac{col}{4} \right\rfloor, \quad n \in \{0,1,2,3\}. \tag{6}$$

The schedule enforces a 12-column spacing between adjacent arrays within the same cycle, yielding a block-domain offset

$$block\_offset = \frac{12}{4} = 3. \tag{7}$$

Therefore, the four arrays request

$$block\_id,\ block\_id+3,\ block\_id+6,\ block\_id+9 \tag{8}$$

within a cycle. We adopt the modulo-unrolling mapping

$$\boxed{bank\_id = (block\_id + 3n) \bmod N_{\text{bank}}} \tag{9}$$

which provides periodic, hardware-friendly addressing while preserving the fixed inter-array spacing property for conflict analysis.

Single-port design by increasing the bank count: For single-port banks, eliminating space-conflict requires pairwise distinct bank IDs in the same cycle, i.e.,

$$3(n_1 - n_2) \not\equiv 0 \pmod{N_{\text{bank}}}, \quad \forall n_1 \neq n_2. \tag{10}$$

With four arrays and $block\_offset = 3$, selecting $N_{\text{bank}} = 12$ satisfies this constraint robustly and yields a regular round-robin placement. Each bank stores every 12th 4-column block in a round-robin interleaving, and The runtime timeline is provided in Appendix A (Table XI), demonstrating that after warm-up four distinct banks are activated per cycle.

Dual-port design by exploiting two reads per bank per cycle: Dual-port banks relax the same-cycle constraint to "no more than two reads per bank per cycle," while still requiring temporal safety against the effective latency L. With $N_{\text{bank}} = 6$, the per-cycle accesses collapse into two active banks per cycle, each serving exactly two reads through deterministic array pairing. Blocks are interleaved round-robin across six banks, and Table XII in Appendix A shows the bank-pair rotation $\{0,3\} \to \{1,4\} \to \{2,5\}$. The reuse distance in this rotation is two cycles; therefore, the schedule is temporally safe when $L \leq 2$.

BRAM utilization and power implications: Table IX shows the single-port design uses 49 BRAM tiles at top level (48 in WBUF), while the dual-port design uses 46 (45 in WBUF) due to reduced bank replication. Power also favors dual-port: single-port consumes 1.60 W total (0.97 W dynamic, 0.62 W static) versus 1.31 W (0.68 W dynamic, 0.62 W

TABLE IX
POST-SYNTHESIS RESOURCE AND POWER COMPARISON BETWEEN SINGLE-PORT AND DUAL-PORT WBUF ORGANIZATIONS.

Metric	Single-port	Dual-port
Bank count	12	6
Top-level BRAM tiles	49	46
WBUF BRAM tiles	48	45
DSP blocks	64	64
Dynamic power (W)	0.971	0.682
Static power (W)	0.626	0.624
Total on-chip power (W)	1.597	1.306

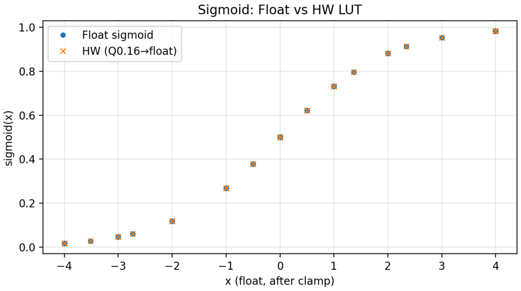

Fig. 6. Sigmoid approximation accuracy: floating-point sigmoid vs. HW LUT outputs.

static), consistent with lower memory/interconnect switching. Therefore, the dual-port WBUF offers a better BRAM–power trade-off without degrading throughput and is selected as the default..

C. Nonlinear Layer Implementation

To realize the sigmoid nonlinearity $\sigma(x) = 1/(1 + e^{-x})$ with low FPGA overhead, we implement a lookup-table (LUT) approximation. The module consumes signed Q8.8 inputs (16-bit) and produces unsigned Q0.16 outputs (16-bit). Inputs are first clamped to $[-4, +4)$. Importantly, this interval is selected based on empirical workload statistics: sampled dt_proj values before the sigmoid are concentrated within this range, which avoids over-provisioning the LUT in saturation regions while keeping the table size compact.

After clamping, the address is generated by a direct fixed-point shift. Since $[-4, +4)$ corresponds to $[-1024, 1023]$ in Q8.8, we compute

$$addr = x_{\text{clamp}} + 1024, \tag{11}$$

to obtain an index in $[0, 2047]$, enabling a 2048-entry LUT that uniformly covers the effective domain. Fig. 6 compares the floating-point sigmoid against the hardware LUT outputs (converted from Q0.16), showing close agreement at the sampled points, thereby validating the chosen clamp range and table resolution.

D. SSM Control and Fine-grained Pipelining

To reduce inter-module coupling and improve scalability, we adopt an AXI-stream-like ready/valid interface and insert lightweight FIFOs between adjacent operators, following the modular streaming philosophy of LightMamba [10]. Under this semantics, a FIFO acts as a standardized boundary for rate matching and timing decoupling. Crucially, the unified interface also simplifies system evolution: extending from an SSM subgraph toward a full Mamba pipeline becomes a matter of composing compliant modules with similar AXI-Stream interfaces, which improves project maintainability and integration robustness.

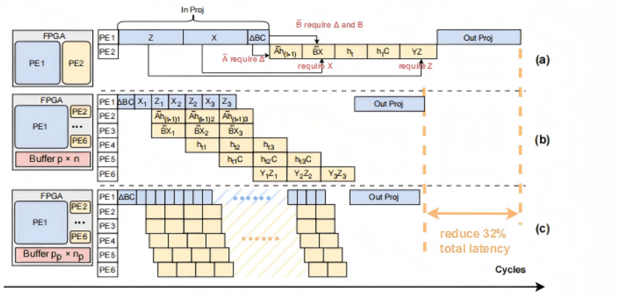

Fig. 7. Fine-grained streaming schedule from LightMamba [10].

Within this control paradigm, we examine the trade-off between reconfigurable and fine-grained pipelines. Many SSM accelerators favor reconfigurable datapaths to reduce resource usage [5]–[7], but increases end-to-end waiting time. In contrast, LightMamba advocates fine-grained streaming, where each MAC tile is immediately forwarded to the sigmoid and element-wise update (EWM/EWA) stages. In our throughput analysis with $N = 64$ tiles, the fine-grained schedule reduces latency from $T_A = 29N$ to $T_B = 22N + 7$, i.e., from 1856 to 1415 cycles (23.8% reduction) while adds only about 12 DSPs per SSM compute path, making the latency gain well justified. Therefore, this project ultimately selects the fine-grained, FIFO-based AXI-stream control scheme.

V. CONCLUSION

From the software perspective, we systematically evaluated Mamba-family variants on LuViRA and identified that short temporal contexts are sufficient for accurate localization. The channel mamba baseline delivers strong accuracy while significantly reducing the model size compared to the prior FCNN, but its multi-branch structure and nonlinear operators complicate quantization and deployment. Motivated by the observed generalization gap and hardware constraints, we proposed slim mamba, which preserves comparable accuracy to channel mamba and the FCNN baseline while substantially reducing training/inference cost and improving deployability. In particular, slim mamba reduces model size to 3.40 MB, which is about 0.77% of the FCNN size (440 MB), and completes training in 13 s per epoch (Table II). We further ablate window length and patch settings and show that a very short window ($K = 1/2$) already yields the best test accuracy, indicating that dense short-term dynamics dominate this task (Table IV).

From a hardware perspective, this work targets the SSM state-update hotspot in Mamba and presents a fully synthesizable FPGA accelerator with an SSM-centric streaming datapath. At 400 MHz, the design closes timing with 76 DSPs and

TABLE X
POST-SYNTHESIS UTILIZATION AND POWER SUMMARY FOR THE INTEGRATED SSM MODULE AT 400 MHZ.

Metric	Value
LUTs	6,244 (2.28%)
FFs	4,928 (0.90%)
BRAM tiles	52 (5.70%)
DSP blocks	76 (3.02%)
Bonded I/Os	136 (41.46%)
Total on-chip power	1.442 W
Dynamic power	0.817 W
Static power	0.625 W
I/O power (dynamic)	0.443 W (54%)
BRAM power (dynamic)	0.089 W
Signals power (dynamic)	0.081 W
Clocks power (dynamic)	0.078 W
Logic power (dynamic)	0.073 W
DSP power (dynamic)	0.053 W

6.2k LUTs (3.0% and 2.3% of the device) and uses 52 BRAM tiles (5.7%) for fully on-chip buffering of streaming MVM and recurrent state updates. Post-synthesis power is estimated at 1.44 W; dynamic power is currently I/O-dominated because the deliverable implements the SSM module without a full-system wrapper. Future work will integrate the module into an end-to-end Mamba pipeline to keep data on-chip and re-estimate power with realistic switching activity.

Overall, the results show that an SSM-centric accelerator can achieve high frequency on FPGA with modest resources. Key contributions include: (i) a primitive-based mapping (MAC/EWA/EWM) that simplifies control; (ii) a pipelined MVM backbone with structured reduction; (iii) an interleaving memory access method; and (iv) LUT-based sigmoid plus FIFO/ready–valid streaming for robust composition and back-pressure handling.

VI. REFLECTIONS ON THIS COURSE

This course trained us in practical research methodology: surveying and critiquing related work, forming testable hypotheses, and iterating on design choices through evidence and discussion. Starting from a baseline plan to implement the Mamba SSM hardware module, we went beyond the initial target by completing the SSM design and analyzing MVM computation, memory-access strategies, and system-architecture trade-offs. The resulting solution is tailored to indoor localization and documented in a public GitHub repository and this report for reproducibility. Participating in the IES Group Meeting also provided firsthand experience of how research is conducted and communicated within the department.

APPENDIX A
BANK ACCESS TIMELINES

TABLE XI
TIMELINE OF BANK ACCESSES FOR $N_{\text{BANK}} = 12$ (SINGLE-PORT).

Cycle	A1→bank	A2→bank	A3→bank	A4→bank	Banks Active
1	bank0	–	–	–	{0}
2	bank4	bank1	–	–	{4,1}
3	bank8	bank5	bank2	–	{8,5,2}
4	bank0	bank9	bank6	bank3	{0,9,6,3}
5	bank4	bank10	bank7	bank1	{4,10,7,1}
6	bank8	bank11	bank2	bank5	{8,11,2,5}
7	bank0	bank3	bank6	bank9	{0,3,6,9}
8	bank4	bank7	bank10	bank1	{4,7,10,1}
9	bank8	bank11	bank2	bank5	{8,11,2,5}
10	bank0	bank3	bank6	bank9	{0,3,6,9}
11	bank4	bank7	bank10	bank1	{4,7,10,1}
12	bank8	bank11	bank2	bank5	{8,11,2,5}

TABLE XII
TIMELINE OF BANK ACCESSES FOR $N_{\text{BANK}} = 6$ (DUAL-PORT).

Cycle	A1→bank	A2→bank	A3→bank	A4→bank	Banks Active
1	bank0	–	–	–	{0}
2	bank4	bank1	–	–	{1,4}
3	bank2	bank5	bank2	–	{2,5}
4	bank0	bank3	bank0	bank3	{0,3}
5	bank4	bank1	bank4	bank1	{1,4}
6	bank2	bank5	bank2	bank5	{2,5}
7	bank0	bank3	bank0	bank3	{0,3}
8	bank4	bank1	bank4	bank1	{1,4}
9	bank2	bank5	bank2	bank5	{2,5}
10	bank0	bank3	bank0	bank3	{0,3}
11	bank4	bank1	bank4	bank1	{1,4}
12	bank2	bank5	bank2	bank5	{2,5}

REFERENCES

[1] I. Yaman, G. Tian, E. Tegler, J. Gulin, N. Challa, F. Tufvesson, O. Edfors, K. Åström, S. Malkowsky, and L. Liu, “Luvira dataset validation and discussion: Comparing vision, radio, and audio sensors for indoor localization,” *IEEE Journal of Indoor and Seamless Positioning and Navigation*, 2024.

[2] G. Tian, I. Yaman, M. Sandra, X. Cai, L. Liu, and F. Tufvesson, “Deep-learning-based high-precision localization with massive mimo,” *IEEE Transactions on Machine Learning in Communications and Networking*, vol. 2, pp. 19–33, 2023.

[3] A. Gu and T. Dao, “Mamba: Linear-time sequence modeling with selective state spaces,” in *First conference on language modeling*, 2024.

[4] C. Zeng, Z. Liu, G. Zheng, and L. Kong, “Cmamba: Channel correlation enhanced state space models for multivariate time series forecasting,” *arXiv preprint arXiv:2406.05316*, 2024.

[5] A. Wang, H. Shao, S. Ma, and Z. Wang, “Fastmamba: A high-speed and efficient mamba accelerator on fpga with accurate quantization,” *arXiv preprint arXiv:2505.18975*, 2025.

[6] J. Li, S. Huang, J. Xu, J. Liu, L. Ding, N. Xu, and G. Dai, “Marca: Mamba accelerator with reconfigurable architecture,” in *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, 2024, pp. 1–9.

[7] X. Jin, H. Zheng, M. Nie, J. Wang, and C.-J. R. Shi, “Hcsas: Hybrid computing systolic arrays for accelerating mamba models with unified state space buffers and energy-efficient dataflow,” in *Proceedings of the Great Lakes Symposium on VLSI 2025*, 2025, pp. 457–462.

[8] G. Tian, I. Yaman, M. Sandra, X. Cai, L. Liu, and F. Tufvesson, “High-precision machine-learning based indoor localization with massive mimo system,” in *ICC 2023-IEEE International Conference on Communications*. IEEE, 2023, pp. 3690–3695.

[9] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal, “Memory bank disambiguation using modulo unrolling for raw machines,” in *Proceedings. Fifth International Conference on High Performance Computing (Cat. No. 98EX238)*. IEEE, 1998, pp. 212–220.

[10] R. Wei, S. Xu, L. Zhong, Z. Yang, Q. Guo, Y. Wang, R. Wang, and M. Li, “Lightmamba: Efficient mamba acceleration on fpga with quantization and hardware co-design,” in *2025 Design, Automation & Test in Europe Conference (DATE)*. IEEE, 2025, pp. 1–7.