

# FastMamba: A High-Speed and Efficient Mamba Accelerator on FPGA with Accurate Quantization

Aotao Wang<sup>1</sup>, Haikuo Shao<sup>1</sup>, Shaobo Ma<sup>1</sup>, and Zhongfeng Wang<sup>1,2</sup>

<sup>1</sup>School of Electronic Science and Engineering, Nanjing University, Nanjing, China

<sup>2</sup>School of Integrated Circuits, Sun Yat-sen University, Shenzhen, China

Email: {atwang, hkshao, shaoboma}@smail.nju.edu.cn, zfwang@nju.edu.cn

**Abstract**—State Space Models (SSMs), like recent Mamba2, have achieved remarkable performance and received extensive attention. However, deploying Mamba2 on resource-constrained edge devices encounters many problems: severe outliers within the linear layer challenging the quantization, diverse and irregular element-wise tensor operations, and hardware-unfriendly nonlinear functions in the SSM block. To address these issues, this paper presents FastMamba, a dedicated accelerator on FPGA with hardware-algorithm co-design to promote the deployment efficiency of Mamba2. Specifically, we successfully achieve 8-bit quantization for linear layers through Hadamard transformation to eliminate outliers. Moreover, a hardware-friendly and fine-grained power-of-two quantization framework is presented for the SSM block and convolution layer, and a first-order linear approximation is developed to optimize the nonlinear functions. Based on the accurate algorithm quantization, we propose an accelerator that integrates parallel vector processing units, pipelined execution dataflow, and an efficient SSM Nonlinear Approximation Unit, which enhances computational efficiency and reduces hardware complexity. Finally, we evaluate FastMamba on Xilinx VC709 FPGA. For the input prefill task on Mamba2-130M, FastMamba achieves  $68.80\times$  and  $8.90\times$  speedup over Intel Xeon 4210R CPU and NVIDIA RTX 3090 GPU, respectively. In the output decode experiment with Mamba2-2.7B, FastMamba attains  $1.65\times$  higher energy efficiency than RTX 3090 GPU.

**Index Terms**—Mamba, quantization, nonlinear optimization, algorithm-hardware co-design, FPGA acceleration.

## I. INTRODUCTION

Mamba [1], as a novel class of State Space Model (SSM), has become a prominent foundation model architecture and has been applied in various deep learning fields such as natural language processing [1], [2], images analysis [3], [4], and video processing [5]. It effectively addresses the quadratic growth in computational complexity of traditional Transformer models [6] concerning input sequence length, offering a more efficient solution for processing longer sequences. Recent Mamba2 [2] has further reduced computational complexity while improving accuracy through state space duality (SSD) framework and semiseparable matrix decompositions and is  $2\sim 8\times$  faster than Mamba.

Deploying models on personal edge devices, rather than cloud servers like the Graphics Processing Unit (GPU), has

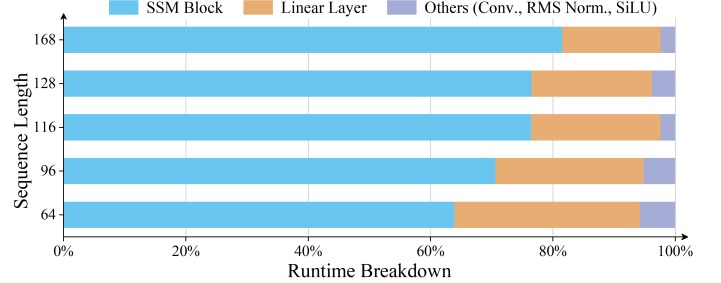


Fig. 1. Runtime breakdown with different evaluation sequence lengths during the prompt prefill stage.

attracted continuous attention for real-time processing and data privacy considerations. Despite Mamba2's advantages, running it on edge hardware like the Field-Programmable Gate Array (FPGA) still faces significant design challenges due to its rigorous computation and memory requirements. Existing algorithm optimization methods, such as quantization [7], [8], are primarily focused on large models based on Transformer architecture, with limited exploration in the context of Mamba. Previously dedicated Mamba ASIC accelerator [9] also lacks sufficient application of quantization methods, which limits its hardware efficiency and flexibility.

The Mamba2 architecture consists of the linear layer, convolution layer, SSM block, and nonlinear functions including Root Mean Square (RMS) normalization [10] and *SiLU* activation [11]. To identify performance bottlenecks, we profiled the runtime of each component in the prompt prefill experiment of Mamba2-130M on RTX 3090 GPU. As shown in Fig. 1, the SSM block and linear layer occupy a majority of the computational load. The runtime proportion of the SSM also increases with the growth of input sequence length. Various kinds of element-wise tensor operations (e.g. add, multiplication, and nonlinear functions) and massive matrix multiplications between activation values and weights [12], constitute the primary computations in the SSM block and the linear layer, respectively. For edge deployment of Mamba2, the dominant SSM block and linear layer face the following challenges: (1) The activation values and weights in the linear layer exhibit severe outlier distributions. Conventional quantization methods fail to mitigate the impact of outlier characteristics, resulting in significant accuracy degradation. Moreover, the nonlinear operations used in SSM block, such as the exponential and *SoftPlus* [13] functions, are sensitive to quantization,

This work was supported in part by the National Key R&D Program of China under Grant 2022YFB4400600, and in part by the Postgraduate Research & Practice Innovation Program of Jiangsu Province under Grant KYCX24\_0149. (Corresponding author: Zhongfeng Wang.)

979-8-3315-3477-6/25/\$31.00 ©2025 IEEE

necessitating carefully designed schemes to maintain accuracy. (2) Diverse and irregular element-wise operations in the SSM block constitute the majority of computational load, presenting the bottleneck to acceleration. So it's necessary to design an efficient hardware architecture to unify various computations. The hardware-unfriendly nonlinear functions inside the SSM block also increase hardware design complexities.

In this paper, we propose FastMamba, an efficient Mamba accelerator with algorithm and hardware co-design to address the above challenges. Our main contributions are as follows:

**(1) Accurate quantization of the linear layer, convolution layer, and SSM block.** We successfully quantize linear layers to 8-bit through Hadamard transformation [8], which eliminates outlier distributions in the linear layer. A hardware-friendly and fine-grained power-of-two (PoT) quantization framework is presented for the convolution layer and SSM block. The quantized algorithm significantly reduces the computational complexity with accuracy degradation within 1%.

**(2) Efficient accelerator design.** We present parallel vector processing units (VPUs) executing basic computations in Mamba2. Based on VPUs, we design efficient hardware for the SSM block and linear layer through group parallelism and pipelined design. Moreover, within the SSM block, we unify *SoftPlus* and exponential functions into hardware-friendly and efficient linear approximation and further design a multi-mode dedicated unit, which effectively saves Digital Signal Processor (DSP) and flip-flop resources.

**(3) Experimental evaluations.** The accelerator is implemented on the Xilinx Virtex-7 VC709 FPGA. For the prompt prefill task on Mamba2-130M, FastMamba achieves a maximum speedup of  $68.80\times$  and  $8.90\times$  compared to the Intel Xeon 4210R Central Processing Unit (CPU) and NVIDIA RTX 3090 GPU, respectively. In the decode experiment with Mamba2-2.7B, the design achieves a  $1.65\times$  improvement in energy efficiency compared to the RTX 3090 GPU.

## II. BACKGROUND

### A. State Space Model

The SSM is a mathematical framework that uses hidden parameters, called “states,” to capture temporal correlations and data relationships. The classical continuous-time SSM, as shown in Eq. (1), consists of a state equation and an output equation. These equations describe the mapping from the input signal  $x(t) \in \mathbb{R}$  to the output signal  $y(t) \in \mathbb{R}$  at time  $t$ , mediated by the hidden state  $h(t) \in \mathbb{R}^N$ :

$$\begin{aligned} h'(t) &= Ah(t) + Bx(t) \\ y(t) &= Ch(t) + Dx(t) \end{aligned} \quad (1)$$

here,  $h'(t) \in \mathbb{R}^N$  is the time derivative of  $h(t) \in \mathbb{R}^N$ , indicating how  $h(t)$  changes over time. The key parameters are the state-transition matrix  $A \in \mathbb{R}^{N \times N}$ , the input matrix  $B \in \mathbb{R}^{N \times 1}$ , the output matrix  $C \in \mathbb{R}^{N \times 1}$ , and the feedthrough matrix  $D \in \mathbb{R}$ .

In practical machine-learning scenarios, data is usually discrete. Thus, discretizing the continuous-time Eq. (1) is

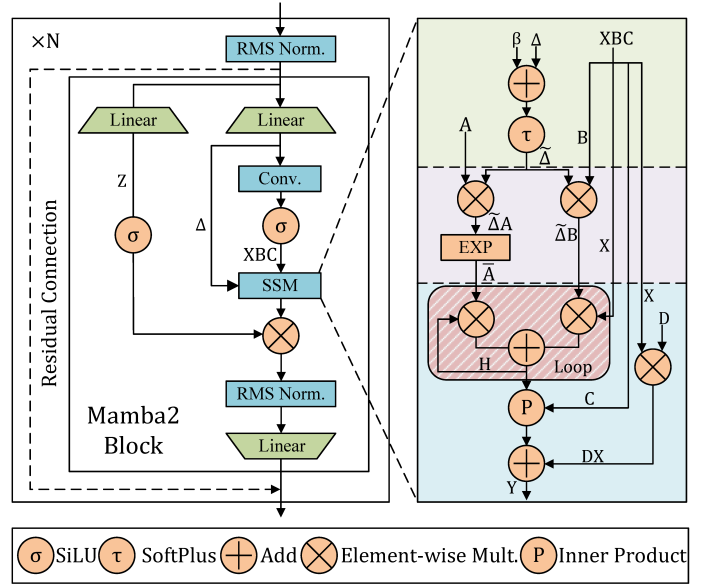


Fig. 2. Computational flow in Mamba2 block and SSM block.

essential. Discretization aims to convert continuous-time parameters like  $A$  and  $B$  into discrete-time parameters. The Zero-Order Hold (ZOH) method [14] is a common approach in the discretization process. It introduces  $\Delta$ , the time interval during which the function value is held constant. Applying the ZOH method transforms the continuous-time Eq. (1) into the following discrete-time Eq. (2):

$$\begin{aligned} h_k &= \bar{A}h_{k-1} + \bar{B}x_k \\ y_k &= Ch_k + Dx_k \end{aligned} \quad (2)$$

where  $\bar{A} = \exp(\Delta A)$ ,  $\bar{B} = (\Delta A)^{-1}(\exp(\Delta A) - I) \cdot \Delta B \approx \Delta B$ , and  $\Delta$  is the discretization step size. In a discrete-time system,  $\bar{A}$  and  $\bar{B}$  are calculated based on  $\Delta$ .

### B. Mamba

The overall computational process of Mamba2 is depicted in Fig. 2. Each layer in Mamba2 consists of 2 RMS normalization layers, multiple linear layers, a convolution layer, 2 *SiLU* (denoted as  $\sigma$ ), an SSM block, and a residual connection.

The input to the SSM block is the data from the *Silu*, split into  $X$ ,  $B$ , and  $C$  along the feature dimension and the  $\Delta$  from the linear layer. The computational flow of the SSM block is partitioned for hardware optimization. Firstly,  $\Delta$  undergoes a preprocessing operation to generate  $\tilde{\Delta}$ . Then,  $\tilde{\Delta}$  is processed with  $A$  and  $B$ , producing  $\tilde{A}$  and  $\tilde{B}$  for subsequent iterations. Finally, after  $L$  (sequence length) iterations, the current  $L$ -time  $H$  performs the inner product operation with  $C$ , and the result is added to the bypass result  $DX$  to produce the final output.

In a real-world deployment, Mamba2 involves two key processes: prompt prefill and token decode. Prompt prefill requires computing with the entire input sequence as context, resulting in intensive GPU memory consumption. In contrast, token decode generates one token by one token. This characteristic enables the decode stage to support inference of large-scale Mamba2 models even with limited hardware resources.

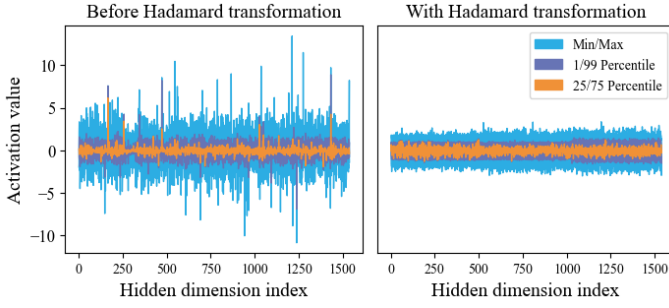


Fig. 3. The distributions of activation values before Hadamard transforming and with Hadamard transforming.

### III. ALGORITHM QUANTIZATION

#### A. Hadamard-based Linear Quantization

In Mamba2, activation values in linear layers typically exhibit more extreme outlier distributions than weights, making their quantization much more difficult. The impact of outliers in activation values  $X$  and weights  $W$  can be weakened by the Hadamard quantization method [8] based on the equation of  $nY = (XH)(H^TW^T)$ , which Hadamard matrix  $H \in R^{n \times n}$  and satisfies  $n = 2^k (k \in \mathbf{Z})$ . Specifically, when normalized by  $\sqrt{n}$ , the Hadamard matrix becomes orthonormal. Therefore, Hadamard transformation is essentially an orthonormal transformation. As shown in Fig. 3, the activation values exhibit a more concentrated distribution with a narrower dynamic range following the Hadamard transformation.

This transformation improves the stability of low-bit quantization. The proposed Hadamard-based Linear Quantization Method (Algorithm 1) uses the special properties of the Hadamard transformation for the 8-bit quantization of activation values and weights.

The activation matrix  $\mathbf{X}$  and the weight matrix  $\mathbf{W}$  are uniformly partitioned into  $m$  groups of sub-matrices  $\mathbf{X}[i]$  and  $\mathbf{W}[i]$  to ensure dimension  $\frac{d}{m} = 2^k (k \in \mathbf{Z})$ . The Hadamard matrix  $\mathbf{H}[i]$  is constructed according to the dimension of  $\mathbf{X}[i]$ . Subsequently, the scaling factors  $s_X$  and  $s_W$  are computed based on the maximum values of  $\mathbf{X}_H$  and  $\mathbf{W}_H$ , and these scaling factors are then utilized to generate the 8-bit  $\hat{\mathbf{X}}_H[i]$  and  $\hat{\mathbf{W}}_H[i]$  within the range of  $-128 \sim 127$ . Ultimately, the partial sums of each group  $\hat{\mathbf{Y}}$  are reduced, and the result is de-quantized to output the result  $\mathbf{Y}$  of the linear layer.

#### B. SSM Quantization and Nonlinear Approximation

PoT quantization represents a hardware-efficient method for converting floating-point to fixed-point by applying a scaling factor of  $2^p (p \in \mathbf{Z})$ . As illustrated in the right section of Fig. 2, the linear operations like add, element-wise multiplication, and inner products are quantified by PoT. However, the PoT quantization can't be directly applied to the nonlinear operations including *SoftPlus* activation and exponential function. Therefore, we optimize these nonlinear functions within a first-order linear approximation algorithm framework aimed at supporting fixed-point quantization by PoT and improving computing efficiency.

#### Algorithm 1 Hadamard-based Linear Quantization Method

**Input:**  $\mathbf{X} \in \mathbb{R}^{l \times d}, \mathbf{W} \in \mathbb{R}^{q \times d}$   
**Output:**  $\mathbf{Y} \in \mathbb{R}^{l \times q}$

- 1:  $\mathbf{X}[i] \in \mathbb{R}^{l \times \frac{d}{m}} \leftarrow \mathbf{X} \in \mathbb{R}^{l \times d}, i = 0, 1, \dots, m-1$
- 2:  $\mathbf{W}[i] \in \mathbb{R}^{q \times \frac{d}{m}} \leftarrow \mathbf{W} \in \mathbb{R}^{q \times d}, i = 0, 1, \dots, m-1$
- 3: **for**  $i = 0, 1, \dots, m-1$  **do**
- 4:  $\mathbf{H}[i] \in \mathbb{R}^{\frac{d}{m} \times \frac{d}{m}} \leftarrow \text{FindHadamard}(\mathbf{X}[i])$
- 5:  $\mathbf{X}_H[i] = \mathbf{X}[i]\mathbf{H}[i]$
- 6:  $\mathbf{W}_H[i] = \mathbf{H}^T[i]\mathbf{W}[i]$
- 7:  $s_X = \text{FindScale}(\text{cat}(\mathbf{X}_H[0], \dots, \mathbf{X}_H[m-1]))$
- 8:  $s_W = \text{FindScale}(\text{cat}(\mathbf{W}_H[0], \dots, \mathbf{W}_H[m-1]))$
- 9: **for**  $i = 0, 1, \dots, m-1$  **do**
- 10:  $\hat{\mathbf{X}}_H[i] = \text{Quant}(\mathbf{X}_H[i], s_X)$
- 11:  $\hat{\mathbf{W}}_H[i] = \text{Quant}(\mathbf{W}_H[i], s_W)$
- 12:  $\hat{\mathbf{Y}} = \hat{\mathbf{Y}} + \hat{\mathbf{X}}_H[i]\hat{\mathbf{W}}_H[i]$
- 13:  $\mathbf{Y} = \hat{\mathbf{Y}} \cdot s_X s_W \cdot \frac{m}{d}$

1) *Exponential linear approximation:* Statistical analysis reveals that all values of the  $\Delta$  tensor are less than 0. Thus, we employ an exponential linear approximation algorithm tailored for the negative-number domain [15]. The algorithm formula is:

$$e^x = 2^{x \log_2 e}, \quad x \leq 0, \log_2 e \approx (1.0111)_2$$

$$= 2^{(u)+(v)} = 2^u \cdot 2^v = 2^v \gg |u|, \quad u, v \leq 0. \quad (3)$$

Here,  $u$  and  $v$  are the integer and fractional part of  $x \log_2 e$  respectively, and  $\gg$  denotes a shift operation. Since  $v \in (-1, 0]$ , we perform an 8-segment high-precision first-order linear approximation of  $2^v$ .

2) *SoftPlus Symmetry Algorithm:* The *SoftPlus* activation function has the following symmetric property [16]:

$$\text{SoftPlus}(x) = x + \text{SoftPlus}(-x) \quad (4)$$

We approximate  $\ln(x)$  using the following first-order linear approximation:

$$\text{SoftPlus}(x) = \ln(1 + e^x) \approx e^x \quad (5)$$

Combining Eq. (3) and the symmetric property of *SoftPlus*, we transform *SoftPlus* into the following system of equations:

$$\text{SoftPlus}(x) = \begin{cases} e^x, & x \leq 0, \\ e^{-x} + x, & x > 0. \end{cases} \quad (6)$$

Through these transformations, for inputs greater than 0, the result of *SoftPlus* can also be calculated using Eq. (3). Ultimately, the nonlinear functions in SSM are unified within the framework of first-order linear calculations. The hardware implementation of *SoftPlus* can reuse the approximate unit of the exponential function, saving hardware resources and reducing computational complexity.

### IV. HARDWARE ARCHITECTURE

#### A. Architecture Overview

The overall architecture of FastMamba is shown in Fig. 4, which consists of the fixed-point computing group (Hadamard-

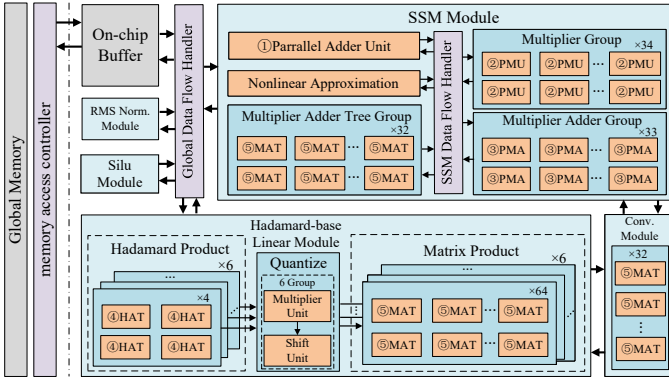


Fig. 4. Overall Architecture of FastMamba.

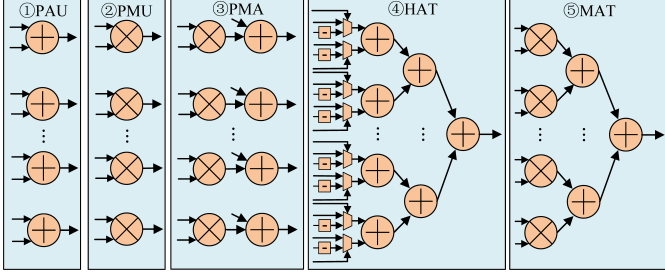


Fig. 5. The structure of the multipliers and adders in the VPUs.

based Linear module, Convolution Module, and SSM Module), as well as the floating-point computing group (RMS Normalization Module and SiLU Module). The data is loaded from Global Memory into the On-chip Buffer. The dataflow between functional modules and buffers is managed by the Data Flow Handler.

In FastMamba, computations can be categorized into addition, multiplication, multiply-add, accumulation, and multiply-accumulate. To handle these operations, we propose VPUs consisting of multipliers and adders as presented in Fig. 5, which involve five types: ① Parallel Adder Unit (PAU), ② Parallel Multiplier Unit (PMU), ③ Parallel Multiplier Adder Unit (PMA), ④ Hadamard Adder Tree (HAT), and ⑤ Multiplier Adder Tree (MAT). The outputs of PAU, PMU, and PMA are vectors of the same length as the inputs, while the outputs of HAT and MAT are reduced to scalars. The inputs, outputs, and corresponding computational functions of each VPU are listed in Table I.

TABLE I: Function Configuration for VPUs

Number	VPU	Input0:Length	Input1:Length	Input2:Length	Output:Length	Function
①	PAU	$A : n$	$B : n$	—	$P : n$	$A + B = P$
②	PMU	$A : n$	$B : n$	—	$P : n$	$A \times B = P$
③	PMA	$A : n$	$B : n$	$C : n$	$P : n$	$A \times B + C = P$
④	HAT	$A : n$	—	—	$P : 1$	$\sum_{i=1}^n (A_i) = P$
⑤	MAT	$A : n$	$B : n$	—	$P : 1$	$\sum_{i=1}^n (A_i \times B_i) = P$

Based on VPUs, the overview of functional modules in different regions of FastMamba is as follows:

**Hadamard-base Linear Module:** This module is designed with 6 parallel computing groups. Each group is designed to perform the Hadamard product, quantization, and 8-bit matrix multiplication.

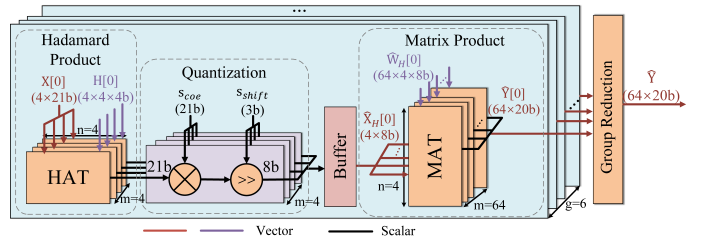


Fig. 6. The architecture of Hadamard-base Linear Module ( $4 \times 21b$  indicates 4 21-bit data).

**Convolution Module:** This module incorporates 32 MAT units to perform 1-D convolution. For a 1-D convolution with a kernel size of 4, each MAT performs the convolution operation on a vector length of 4.

**SSM Module:** Various VPUs are employed in this module to perform fixed-point computations required by SSM block. A Nonlinear Approximation Unit is also designed to compute exponential and *SoftPlus* functions with 16-bit fixed-point data based on our algorithm approximation.

**Floating-point Modules:** As demonstrated in Fig. 1, the RMS normalization and *SiLU* activation contribute a relatively small proportion in Mamba2 computational load. Utilizing floating-point computing units can effectively avoid accuracy loss with slight hardware overhead.

### B. Hadamard-based Linear Module

The Hadamard-based Linear Module is designed with 6 computing groups (as shown in Fig. 6), where each group indexed by  $i = 0, \dots, 5$ , can fully execute the linear quantization computation and output partial sums.

In the Hadamard product, 4 parallel HAT units share the input activation values  $X[i]$  and respectively receive the Hadamard matrix  $H[i]$  to compute and output 4 scalar intermediate values. These intermediate values go through multiplication ( $\times s_{coe}$ ) and shifting ( $\gg s_{shift}$ ) operations for hardware-friendly quantization, and are then spliced along the feature dimension to form an 8-bit quantized activation vector  $\hat{X}_H[i]$  of length 4, which is temporarily stored in a buffer.

This module also has 64 parallel MAT units for the matrix product. These units share the input  $\hat{X}_H[i]$  and respectively receive 8-bit weights  $\hat{W}_H[i]$  to output partial sums  $\hat{Y}[i]$  of length 64. The partial sums of each group are reduced to generate the linear output  $\hat{Y}$ .

### C. SSM Module

Following the three-step computations corresponding to Fig. 2, we design the SSM Module as shown in Fig. 7. The specific details are as follows:

**Step1:** We design a PAU and a Nonlinear Approximation Unit, each having an input vector length of 24. In the *SoftPlus* mode configuration of the Nonlinear Approximation Unit, it computes and outputs  $\hat{\Delta} \in R^{1 \times 24}$ .

**Step2:** One of the paths consists of a PMU and a Nonlinear Approximation Unit both with an input vector length of 24. When the Nonlinear Approximation Unit is set to the exponential mode, this design outputs  $\bar{A} \in R^{1 \times 24}$ . Besides, a PMU with an input vector length of 64 computes  $Q \in R^{1 \times 64}$ .



TABLE II: Comparison of Perplexity and Accuracy of different quantization algorithms on Mamba2-130M.

Method	Precision	Lambada PPL(↓)	Lambada ACC(↑)	Hellaswag ACC(↑)	Piqa ACC(↑)	Arc-easy ACC(↑)	Arc-challenge ACC(↑)	Winogrande ACC(↑)	Openbookqa ACC(↑)	Average ACC(↑)
NormalQ	W8A8	33.7	32.5	33.9	62.3	44.5	23.6	50.9	30.6	39.8
SmoothQ [7]	W8A8	19.1	42.9	34.8	64.1	46.6	23.8	52.0	27.6	41.7
<b>FastMamba-LQ</b>	W8A8	17.2	43.1	35.3	64.8	47.2	24.5	53.1	30.4	<b>42.6</b>
<b>FastMamba</b>	W8A8	17.9	43.0	34.8	64.7	47.0	23.6	52.6	29.6	42.2
Mamba2-130M	FP16	16.9	43.9	35.3	64.9	47.4	24.2	52.1	30.6	<b>42.6</b>

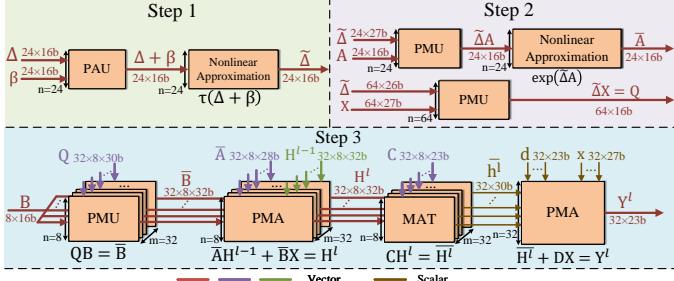


Fig. 7. Architecture and computational flow of the SSM module (24×16b indicates 24 16-bit data).

**Step3:** Initially, we set up 32-parallel PMU and PMA units to generate hidden-state vectors  $H^l \in R^{32 \times 8}$  for the current  $l$ . Then, the  $H^l$  undergoes inner product processing with  $C$  by 32-parallel MAT, outputting the scalars  $h^l$ . Finally, the scalars of  $D$  ( $d$ ), the scalars of  $X$  ( $x$ ) and  $h^l$  are fed into a 32-input PMA unit to perform linear computation generating the SSM output  $Y^l$ .

#### D. Nonlinear Approximation Unit in SSM

As depicted in Fig. 8, we propose a 24-parallel Nonlinear Approximation Unit that can compute both the exponential function and *SoftPlus* activation. It supports the 16-bit fixed-point vector with a length of 24 both inputs and outputs. The EXP-INT part fully implements the Eq. (3). When the input scalar  $x_i \leq 0$ , the exponential mode is activated. In this mode,  $x_i$  directly traverses the EXP-INT part to obtain the exponential output  $\exp(x_i)$ . The exponential mode is also capable of computing the *SoftPlus* when the  $x_i \leq 0$ .

When  $x_i > 0$ , the *SoftPlus* mode is set. First, the Reverse Process Unit (RPU) in the Preprocessing part converts  $x_i$  to its negative value  $-x_i$ , while  $x_i$  is temporarily stored in the Delay Unit. Then,  $-x_i$  passes through the EXP-INT part to compute the intermediate result  $\exp(-x_i)$ . Finally,  $\exp(-x_i)$  and  $x_i$  from the Delay Unit are summed by the adder in the Postprocessing part.

### V. EXPERIMENTAL RESULTS

#### A. Experimental Setup

**Models and Datasets.** During the prompt prefill stage, because of the limited GPU memory, we employ the Mamba2-130M model to evaluate the accuracy of our algorithm under 8-bit activation values and weights quantization (denoted as W8A8). We report the perplexity and zero-shot accuracy on 7 tasks: Lambada, Hellaswag, Piqa, Arc-easy, Arc-challenge, Winogrande, and Openbookqa [2] by using the lm-evaluation-harness [17] tool. Moreover, the Mamba2-130M is also utilized to evaluate the speedup ratio of FastMamba. In the

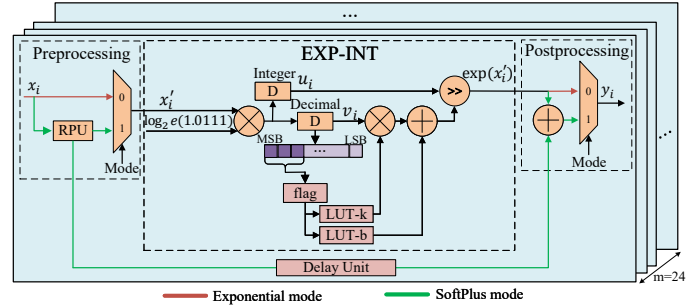


Fig. 8. The multiplex structure of the Nonlinear Approximation Unit

decode stage, we use the Mamba2-2.7B model to evaluate the throughput and energy efficiency of FastMamba.

**Algorithm Evaluation Baselines.** We choose the Mamba2-130M model in the data format of half-precision floating-point (denoted as FP16) as the baseline. We also apply Normal Quantization (NormalQ) and Smooth Quantization (SmoothQ) [7] that perform W8A8 quantization only on linear layers as a comparison. In NormalQ, no optimization is processed for the outliers of activation values and weights.

**Hardware Evaluation Platforms.** We implement FastMamba on the Xilinx Virtex-7 VC709 FPGA. We selected the Intel Xeon Silver 4210R CPU and the NVIDIA RTX 3090 GPU as baseline platforms, denoted as CPU and GPU, respectively. The system configuration of the CPU, GPU, and FPGA are presented in Table III.

TABLE III: System Configuration.

	CPU	GPU	FastMamba
<b>Platform</b>	Intel Xeon Silver 4210R(14nm)	NVIDIA GeForce RTX 3090(8nm)	Xilinx Virtex-7 VX690T(28nm)
<b>Frequency</b>	2400MHz	1395MHz	250MHz
<b>Computing Units</b>	10 Cores	328 Tensor Cores	3333 DSPs
<b>Throughput</b>	-	111 token/s	5.68 token/s
<b>Energy Efficiency</b>	-	0.37 token/(s·W)	0.61 token/(s·W)

#### B. Accuracy Evaluations

Table II presents the accuracy performance of W8A8 quantization using different methods based on Mamba2-130M. In the table, FastMamba-LQ only quantizes the linear layer, while FastMamba quantizes the linear layer, convolution layer, and SSM block.

The results show that FastMamba-LQ surpasses NormalQ and SmoothQ in both perplexity and accuracy. The outlier features with distinct degrees still exist in activation values and weights after NormalQ and SmoothQ processing. In contrast, the Hadamard transformation evenly disperses the outliers of activation values and weights across channels, significantly

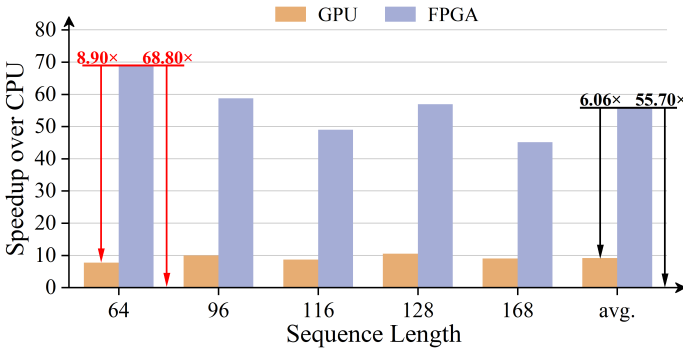


Fig. 9. Comparison of speedup improvement to CPU and GPU on Mamba2-130M with different input sequence length during prompt prefill stage.

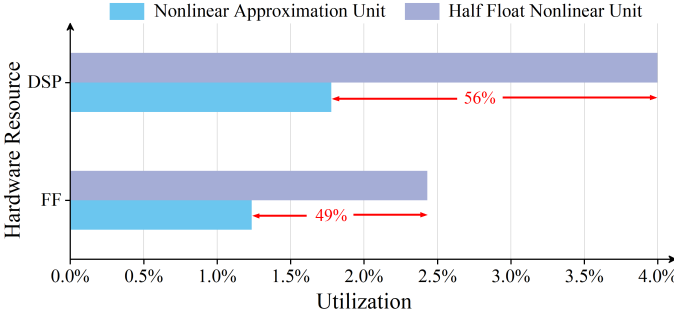


Fig. 10. The hardware resource utilization rates of the Nonlinear Approximation Unit and the Half Float Nonlinear Unit.

TABLE IV: FastMamba FPGA Resources Utilization

Component	LUT	FF	DSP	BRAM
Linear	132030(30.5%)	84514(9.8%)	48 (1.3%)	0
Convolution	14125(3.3%)	13201(1.5%)	256(7.1%)	0
SSM	73597(17.0%)	58196(6.7%)	2376(66.0%)	0
RMS Norm. & SiLU	57315(13.2%)	87633(10.1%)	461(12.8%)	0
Buffer	13597(3.1%)	64898(7.5%)	0	956(65.0%)
Others	44120(10.2%)	46022(5.3%)	192(5.3%)	0
Total	334784(77.3%)	354464(40.9%)	3333(92.5%)	956(65.0%)

mitigating the impact of outlier features and attaining optimal accuracy. Due to applying the PoT quantization in the convolution layer and SSM block, FastMamba experiences a minor accuracy reduction within an acceptable range compared to FastMamba-LQ.

### C. Hardware Evaluations

1) *Prompt Prefill Speedup*: As shown in Fig. 9, during the prompt prefill stage on Mamba2-130M, compared with CPU and GPU, FastMamba achieves a maximum speedup ratio of  $68.80\times/8.90\times$  and an average speedup ratio of  $55.70\times/6.06\times$  at the typical evaluation length. The improvement in acceleration performance mainly originates from fixed-point quantization, parallel VPUs, and pipeline design.

2) *Decode Throughput and Energy Efficiency*: As Mamba2 does not support CPU decode inference, we only report the throughput and energy efficiency of the GPU and FastMamba on Mamba2-2.7B, as shown in Table III. FastMamba exhibits an energy efficiency improvement of  $1.65\times$  compared to the GPU baseline.

3) *Resource report*: Table IV presents the resource consumption of FastMamba. The Hadamard-based linear module

and the SSM module consume the majority of FPGA resources. In the Hadamard-based linear module, the 8-bit MAT units are mainly implemented using Look-Up Table (LUT) units. In the SSM module, as there are the most VPUs, it consumes the largest number of DSP resources. As illustrated in Fig. 10, our Nonlinear Approximation Unit saves 56% of DSP resources and 49% of flip-flop (FF) resources, compared with the Half Float Nonlinear Unit using FP16 data format.

## VI. CONCLUSION

This paper proposes FastMamba, a dedicated accelerator on FPGA for Mamba2 with hardware-algorithm co-design. We achieve 8-bit quantization through Hadamard transformation for the linear layer while applying PoT quantization for the convolution layer and SSM block. To optimize the nonlinear functions in the SSM block, we propose a hardware-friendly and efficient linear approximation and design a dedicated Nonlinear Approximation Unit. Moreover, we implement an accelerator on Xilinx VC709 FPGA utilizing parallel vector processing units and high-efficiency pipelining. For the prefill task on Mamba2-130M, FastMamba achieves a maximum  $68.80\times$  and  $8.90\times$  speedup over the Intel Xeon 4210R CPU and NVIDIA RTX 3090 GPU, respectively. In the output decode experiment with Mamba2-2.7B, our design achieves a  $1.65\times$  higher energy efficiency than the RTX 3090 GPU.

## REFERENCES

- [1] A. Gu *et al.*, “Mamba: Linear-time sequence modeling with selective state spaces,” *arXiv preprint arXiv:2312.00752*, 2024.
- [2] A. Gu *et al.*, “Transformers are SSMs: Generalized Models and Efficient Algorithms Through Structured State Space Duality,” in *ICML*, 2024.
- [3] L. Zhu *et al.*, “Vision Mamba: Efficient Visual Representation Learning with Bidirectional State Space Model,” in *ICML*, 2024.
- [4] Y. Liu *et al.*, “VMamba: Visual State Space Model,” in *NeurIPS*, 2024, pp. 103031-103063.
- [5] K. Li *et al.*, “VideoMamba: State Space Model for Efficient Video Understanding,” in *ECCV*, 2024, pp. 237-255.
- [6] A. Vaswani *et al.*, “Attention is All you Need,” in *NeurIPS*, 2017.
- [7] G. Xiao *et al.*, “SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models,” in *ICML*, 2023.
- [8] S. Ashkboos *et al.*, “QuaRot: Outlier-Free 4-Bit Inference in Rotated LLMs,” in *NeurIPS*, 2024, pp. 100213-100240.
- [9] J. Li *et al.*, “Marca: Mamba accelerator with reconfigurable architecture,” *arXiv preprint arXiv:2409.11440*, 2024.
- [10] B. Zhang *et al.*, “Root Mean Square Layer Normalization,” *arXiv preprint arXiv:1910.07467*, 2019.
- [11] P. Ramachandran *et al.*, “Searching for Activation Functions,” *arXiv preprint arXiv:1710.05941*, 2017.
- [12] M. T. Khan *et al.*, “Digit-Serial DA-Based Fixed-Point RNNs: A Unified Approach for Enhancing Architectural Efficiency,” *IEEE Trans. Neural Networks and Learn. Syst.*, vol. 36, no. 5, pp. 8240-8254, 2025.
- [13] C. Dugas, “Incorporating Second-Order Functional Knowledge for Better Option Pricing,” *Neural Information Processing Systems*, 2000.
- [14] G. Pechlivanidou *et al.*, “Zero-order hold discretization of general state space systems with input delay,” *IMA J. Math. Control. Inf.*, vol. 39, pp. 708-730, June 2022.
- [15] D. Zhu *et al.*, “Efficient Precision-Adjustable Architecture for Softmax Function in Deep Learning,” *IEEE Trans. Circuits Syst.*, vol. 67, pp. 3382-3386, December 2020.
- [16] X. Feng *et al.*, “A High-Precision Flexible Symmetry-Aware Architecture for Element-Wise Activation Functions,” in *ICFPT*, 2021, pp. 1-4.
- [17] L. Gao *et al.*, “A framework for few-shot language model evaluation,” *Zenodo*, July 2024. [Online]. Available: <https://zenodo.org/records/12608602>.