Tensor Parallelism for State-Space Models

Anurag Dutt, Nimit Shah, Hazem Masarani, Anshul Gandhi
PACE Lab, Stony Brook University

***Abstract*—State space models (SSMs) have recently emerged as an effective backbone for sequence modeling and are increasingly used in large language models (LLMs), particularly in long-context regimes. However, serving performance for SSM-based LLMs is constrained by the capabilities of a single GPU, motivating GPU-parallel implementations to scale capacity and improve performance. While tensor parallelism (TP) is a well-established mechanism for scaling LLM inference, it is difficult to employ TP for SSMs due to their distinct structure: they include a sequence-wise recurrent state update and local mixing whose efficiency depends on keeping state and key intermediates local. In this paper, we present a communication-efficient tensor-parallel design and implementation for SSM inference that overcomes the systems challenges of parallelizing SSMs by introducing an SSM cache to reuse per-layer recurrent state and by designing an SSM-aware partitioning to preserve local state updates while avoiding unnecessary synchronization. Our experimental evaluation of four representative SSM-based LLMs (Mamba, Mamba-2, Falcon-Mamba, and Zamba) on two GPU platforms (NVIDIA A6000 and A100) shows that our tensor-parallel SSM inference on 2 and 4 GPU setups improves throughput by $\sim$1.4–3.9$\times$ compared to single-GPU inference. Our design also allows us to handle $\sim$2–4$\times$ larger batch sizes and output sequence lengths due to efficient memory usage compared to Data Parallelism and single-GPU inference.**

***Index Terms*—Efficient Inference, GPU Parallelism, Tensor Parallelism, State Space Models, Mamba, LLMs**

I. Introduction

State Space Models (SSMs) have recently emerged as a strong backbone for sequence modeling, with competitive results and rapid adoption in large language modeling [1]–[4]. A key appeal is their ability to handle long contexts efficiently: instead of forming pairwise token interactions as in the attention mechanism in Transformer models, SSM layers process tokens in order while maintaining a compact internal *state* that summarizes prior context [1], [5]. SSM mixers also scale *linearly* in sequence-processing costs compared to transformers (self-attention), which are *quadratic*. An SSM block processes input tokens by continuously updating the internal state as it reads tokens, and producing an output for each token from this evolving state memory; the way the memory is updated depends on the current token, which makes SSMs flexible and improves modeling quality at scale. We discuss SSMs in detail in Section II-A.

As SSMs move to deployment-relevant model sizes, the systems focus shifts to *improving serving performance*: maximizing throughput under the memory and bandwidth constraints of the deployment. In particular, as SSM models grow and serving stacks support longer contexts and higher concurrency, deployments must simultaneously (i) fit model weights and runtime buffers, and (ii) sustain high token throughput. However, despite their advantages and flexibility, the serving performance of SSMs in practice is largely constrained by the limits of a single GPU (memory and compute capabilities). As SSM model sizes grow, serving performance is increasingly constrained by the ***lack of an established GPU-parallelized implementation***.

The common approach to harnessing the resources of multiple GPUs is through parallel execution of the model across the GPUs. **Tensor parallelism (TP)** is the central mechanism for parallelizing model inference on multi-GPU systems. TP partitions the parameters *and* computation of individual model layers across GPUs, so a single forward pass is executed collaboratively by multiple GPUs, improving throughput [6]. This way, the entire model need not be replicated on each GPU, thus overcoming the memory limitations of a single GPU faced by SSMs [7], [8]. Further, by splitting per-token compute across multiple GPUs, TP also helps to reduce per-request (or per-batch) latency.

Despite the prevalence and support of TP for other models, there is not yet a standard, "drop-in" TP implementation for SSMs and its variants. For Transformers, for example, TP has converged to well-tested patterns because most of the work inside each Transformer block is a small number of large matrix multiplications, and it is relatively clear where to split those computations across GPUs [6]. SSMs are organized differently. Each SSM consists of sequential SSM mixer blocks chained together to transform token embeddings into progressively richer representations, culminating in the final hidden states used to predict the next token. SSMs also include a state-update operator that is applied across the sequence and lightweight local mixing such as convolution [1], [5]. These operations have ***strict locality and memory-layout requirements*** to achieve high throughput, and they interact with intermediate activations that are often larger than the final model-space output. As a result, applying Transformer TP templates, that are designed around sharding General Matrix-Matrix multiplications (GEMMs), can inadvertently place communication at unfavorable points in the mixer, leading to poor scaling.

In practice, designing an effective tensor-parallel SSM implementation is complicated by the following challenges:

- **Reusing per-layer state across phases.** Inference serving naturally has an initial prompt pass followed by token-by-token generation. This requires the internal state of the input prompt to be reprocessed for *every* token generation, significantly increasing latency on the critical path.
- **Communication-aware sharding.** A good TP setup would shard work so each GPU can execute its portion of the mixer block using local tensors, without repeatedly reassembling intermediates. This requires choosing shard boundaries that match how the SSM block produces and consumes interme-

This work was supported by NSF grants CCF-2324859, CNS-2214980, and CNS-2106434.

diate activations. Naively sharding on the first dimension of the local tensor (as is the case in successful Transformer TP implementations) leads to extra synchronization steps.

- **Handling packed parameter blocks.** SSM mixers often pack multiple logical quantities into a single tensor for efficiency, but uniformly slicing this tensor across GPUs can separate fields that must remain together, forcing costly reconstruction and extra communication.
- **Minimizing unavoidable inter-GPU communication.** While some cross-GPU communication is necessary to produce the correct block output under TP, it still imposes significant latency that affects end-to-end throughput.

Prior work has largely focused on algorithmic and kernel efficiency for SSMs on a *single GPU* [1], [5], [9]. There are also early efforts to integrate specific SSM implementations into distributed training stacks on specialized platforms (e.g., Mamba-2 training with neuronx-distributed on AWS Trainium), underscoring practical demand for parallel execution [10]. However, to the best of our knowledge, ***tensor-parallel inference for modern SSMs is not yet a well-established capability*** in mainstream serving stacks as existing TP implementations for LLMs do not extend to SSMs [6], [7].

In this paper, we present a tensor-parallel inference design for selective SSMs that directly addresses the above-mentioned practical challenges that arise in serving. The key contributions of our design are as follows:

1) We introduce a ***distributed caching mechanism*** that prevents reprocessing the input prompt for every newly-generated token by persisting the model's per-layer internal state. The cache contents are carefully selected and sharded across GPUs to retain the locality needed for continued computation without additional inter-GPU communication.
2) We design an ***intelligent tensor-parallel sharding scheme*** that aligns shards with the mixer's computation so that the core SSM-path operations stay GPU-local, avoiding communication introduced by naive splitting.
3) We ***handle the mixer's packed parameters explicitly*** by partitioning the packed block to preserve the fields required for correct state updates, ensuring each GPU has the components it needs for its local update while distributing the remaining storage and compute across GPUs.
4) We reduce the communication overhead of the TP synchronization step by ***introducing quantized communication*** tailored to the SSM's output aggregation during inference.

We experimentally evaluate our TP design on four representative SSM-based LLMs that cover both pure-SSM and hybrid architectures: Mamba, Mamba-2, Falcon-Mamba, and Zamba [1], [5], [11], [12]. Tested on two GPU clusters, NVIDIA A6000 and NVIDIA A100, we show that our efficient TP design ***increases throughput by 1.5–1.9× on 2 GPUs*** and by ***2.4–3.9× on 4 GPUs*** compared to single-GPU inference, across all four SSMs, with the largest gains at long contexts. Importantly, our TP implementation allows us to support ***significantly larger prompts sizes (2–4×)*** compared to single-GPU and data-parallel inference. Furthermore, applying quantized AllReduce for slightly reduced precision improves throughput by an additional 10–18% by lowering synchronization bandwidth overhead.

The rest of this paper is organized as follows. Section II provides background on SSMs and tensor-parallel inference. Section IV presents our key contribution: the tensor-parallel design for SSM inference. Section V evaluates performance and scaling behavior of our tensor-parallel design for four SSMs under two different GPU clusters.

II. Background

In this section, we first discuss State Space Models (SSMs) in more detail, and the necessary background on tensor parallelism. As SSMs have matured, they have become a ***practical alternative to attention modules*** for long-context modeling. Compared to self-attention, which forms token-to-token interactions and therefore incurs *quadratic* work in sequence length and a growing KV-cache during decoding, SSM mixers update a compact recurrent state in a streaming manner in ***linear*** time complexity. As a result, SSMs can offer better long-context efficiency and lower decode-time memory overhead, which is attractive for efficient serving.

Mamba is a prominent selective SSM mixer that enables competitive attention-free language models by combining token-wise projections with efficient state-update computation [1]. Building on this direction, **Falcon-Mamba** demonstrates a *pure* Mamba-based, attention-free long-sequence generation of SSMs through substantially lower memory overhead [11]. In parallel, hybrid architectures have incorporated attention sparingly to recover retrieval and in-context learning behavior while retaining the efficiency of SSM backbones [13]. **Zamba** proposes a compact hybrid model that pairs a Mamba backbone with a single shared self-attention module, aiming to preserve attention's benefits at minimal parameter cost [12]. **Mamba-2** introduces State Space Duality that reformulates the Mamba mixer's state-update computation to map cleanly onto matrix-multiplication style kernels, easing scalability [5]. Together, the development of these model families indicates that SSMs are becoming a standard component of the LLM design space.

A. Overview of SSM architecture

A state-space model (SSM) processes a sequence by maintaining a small internal *state* that summarizes prior context and is updated as new tokens arrive. Instead of computing pairwise interactions between all tokens (as in self-attention), an SSM updates this state over time and produces an output for each token from the evolving state. For practical deployment, this "streaming" structure is appealing because prompt processing ***scales linearly*** with sequence length and token-by-token generation. Modern SSM layers package these ideas into a practical **SSM mixer block** that combines Input projections with a state-update path as shown in Figure 1. We next describe the typical Mamba-style mixer block at a high level in the same order as it is executed.

The first SSM mixer block receives the prompt activations as the residual-stream tensor `hidden_states`. An *input projection* expands this representation into a wider intermediate tensor

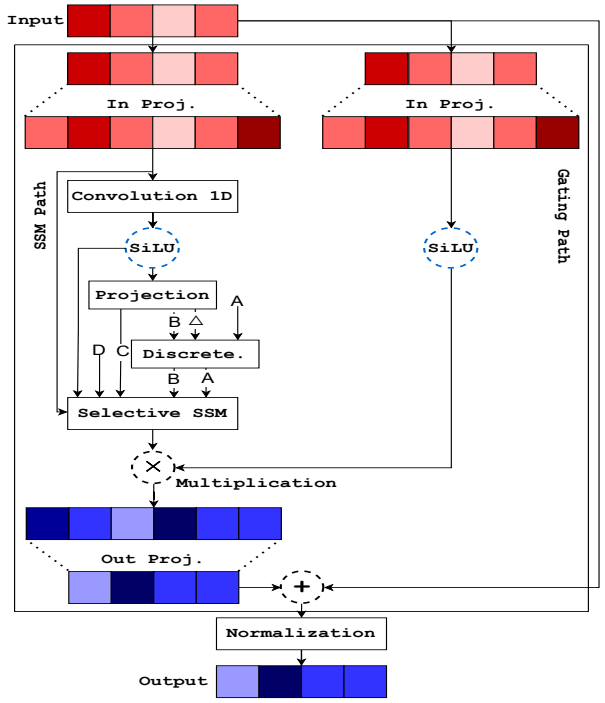


Fig. 1: Mamba-style SSM mixer block. The input projection splits into an SSM path and a gating path; the SSM path applies channel-separable convolution, generates SSM fields (Δ, B, C) with per-channel parameters (A, D), performs the state update, gates the result, and projects back to the residual stream.

and is immediately split into two branches: (i) an *SSM path* that performs sequence modeling via convolution and a state update, and (ii) a *gating path* that produces a multiplicative gate used later to modulate the SSM-path output.

In the SSM path, a short 1D convolution is applied. Next, a lightweight projection maps the convolved activations into the *token-dependent parameter fields* used by the SSM update. These fields are produced in packed form and then split into the logical quantities that drive the update:

- Δ: a per-token *step size* that controls how the continuous dynamics are converted into a discrete update.
- B and C: per-token *input/output* fields; B determines how the current token influences the state update, and C determines how the state is read out into an output for that token.

In addition to these token-dependent fields, the block uses learned per-channel parameters that are fixed for the layer:

- A: per-channel *state dynamics* parameters that define how the state evolves over time.
- D: a per-channel *skip/scale* term that provides a direct contribution from the activation stream to the output (often viewed as a residual-like bypass inside the mixer).

Before applying the recurrence, the block performs a *discretization* step that combines Δ with the fixed dynamics A (and scales B accordingly) to form the coefficients used by the discrete-time update. The *SSM update* (also called the scan/state-update) then walks forward over the sequence: it updates the per-channel recurrent state at each token and emits a per-token output via C, with an additional per-channel contribution via D. In practice, these token-dependent SSM quantities are generated as a single *packed parameter tensor* and then `split` into Δ, B, and C for the state-update computation.

The SSM-path output is then combined with the gating path by elementwise multiplication (the gate controls how much of the SSM output is passed through). Finally, an *output projection* maps the result back to the model hidden size, and the block returns to the residual stream by adding this output to the incoming `hidden_states`.

The same mixer structure also appears in recent model families that use Mamba-style SSM blocks as core building units, such as Mamba-2 Falcon-Mamba and Zamba. From a systems and tensor-parallel perspective, both models still inherit the same key parallelization constraints for the SSM mixer path: intermediate activations are packed and then split into the fields needed by convolution and the SSM update.

B. Tensor parallelism and SSMs

Tensor parallelism (TP) is a model-parallel strategy that splits the parameters and computation of a *single layer* across multiple GPUs so they collaboratively execute one forward pass [6]. In Transformer models, TP is commonly realized by sharding the large projection matrices in attention and linear layers and inserting communication collectives (e.g., AllGather/AllReduce) to form the correct activations.

The operator mix in modern SSM blocks differs materially from standard Transformer blocks. While SSM layers still include token-wise matrix multiplications that resemble Transformer linear projections, they also include sequence-wise kernels (selective scan/state update) whose data dependencies and intermediate-state structure are not identical to attention KV-cache access patterns. Because the SSM mixer's state-update path requires per-channel contiguous intermediate slices (from packed parameter tensors) and local recurrent state updates, naively applying standard Transformer TP templates can fragment these layouts and introduce extra communication on the critical path. Consequently, the TP strategies that work well for Transformers do not necessarily transfer directly to SSMs, and ***must be revisited*** with respect to SSM-specific computation graphs and state layouts. This gap motivates this work on SSM-aware TP design and implementation.

III. Related Work

Prior work on SSM development has proposed hardware and compiler optimizations, including PIM-based acceleration and fine-grained GPU scheduling approaches [14], [15]. There is also work on algorithmic and kernel efficiency improvements for SSMs [1], [5], [9], [11], [12]. However, all of these works are limited to a *single-GPU* SSM deployment. Consequently, the parallelization of SSMs remains underexplored despite the growing popularity of SSM-based model families. To our knowledge, there is no published work on parallel implementations of SSMs. However, there is related work on parallelizing Transformer models and reducing the overheads in multi-GPU settings, which are relevant to our design.

A. Transformer-oriented Tensor Parallelism (TP) techniques

We discuss Transformer-oriented TP because Transformers have been the dominant sequential token-generation workload

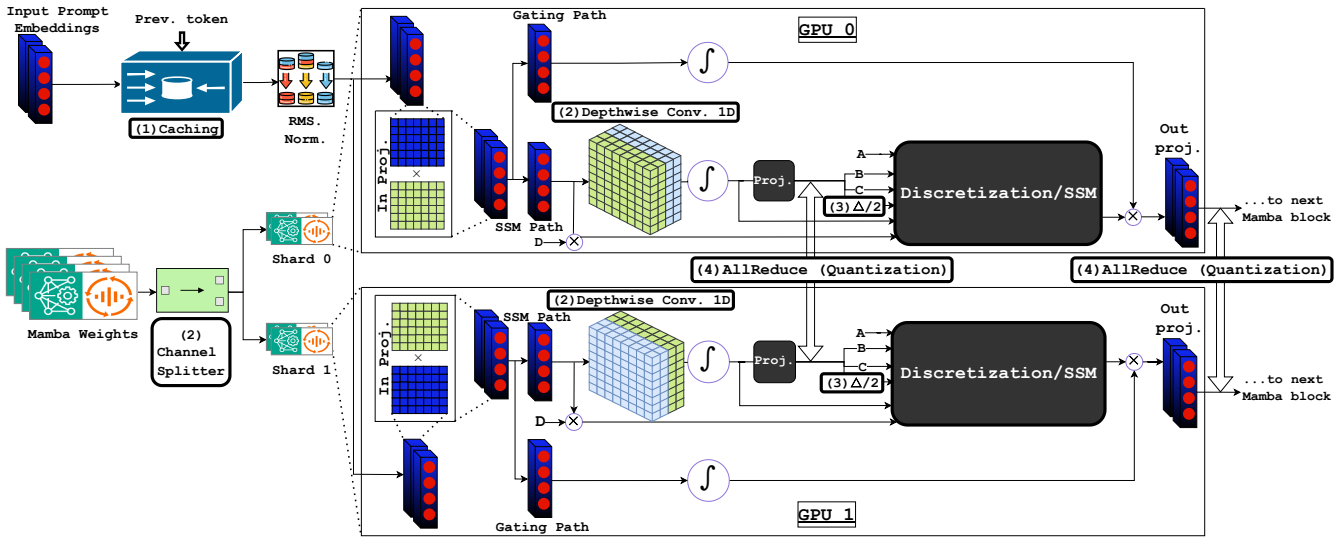


Fig. 2: Illustration of our tensor-parallel inference implementation for Mamba, consisting of our four key design components: (1) SSM cache, (2) channel splitter and depthwise convolution, (3) packed parameter handling, and (4) AllReduce quantization.

in practice, and most mature multi-GPU inference parallelism templates and systems were developed around Transformer blocks—making them the natural starting point (and baseline) when extending TP support to SSM mixers. TP is well established for Transformer models, where compute is dominated by dense matrix multiplications and a small set of communication collectives. Megatron-LM introduced practical column/row-parallel templates for Transformer layers and demonstrated scalable training and inference under TP configurations [6], [16]. DeepSpeed and Colossal-AI provide end-to-end system stacks that integrate TP with memory optimizations, pipeline execution, and distributed runtime support [7], [17]. These systems heavily influenced today's production TP practices. However, as discussed in Section II-A, SSM blocks introduce additional projections and state-space operations that require revisiting sharding and communication placement beyond the Transformer case [1], [5].

B. Communication compression and quantization

A key performance issue with multi-GPU TP designs is the communication latency that is incurred during inter-GPU communication. In general, communication-efficient distributed training and inference has a long history of using compression and quantization techniques to reduce the payload of collective (communication) operations. Communication overhead reduction techniques using low-bit/1-bit and stochastic quantization, as well as low-rank/structured compression, have been revisited by recent studies for optimizing large-scale model parallelism communication collectives [18]–[21].

At the systems level, modern GPU software stacks are also beginning to expose low-precision communication primitives (e.g., FP8 and FP16-oriented support in Transformer Engine), making quantization increasingly practical in end-to-end deployments [22]. Building on these techniques, our TP design for SSM inference includes an optional quantization component, as described in Section IV-D.

IV. Design

This section describes our key contribution: the system design for enabling *tensor-parallel (TP) inference* for SSM-based models. The core challenge is that an SSM mixer is not just a sequence of large matrix multiplications, as in a Transformer; it also contains locality-sensitive operators (notably depthwise convolution and an SSM state-update kernels) that achieve high single-GPU performance by operating on GPU-resident, contiguous tensors with minimal intermediate materialization. A naive reuse of standard TP templates (such as those for Transformers) will ***break locality***, thereby inducing *expensive and unnecessary inter-GPU communication* (AllGathers/AllReduces) and layout transforms at inappropriate locations, which increases memory traffic, breaks fused execution, and reduces throughput.

To make TP practical for SSMs, we make four key design contributions, in the following order, as shown in Figure 2: (1) caching for low latency, (2) intelligent channel-wise model splitting to reduce synchronization and depthwise convolution, (3) handling of SSM parameter streams under sharding, and (4) optional AllReduce quantization. We now discuss these design decisions in detail. We commit to making our inference engine implementation publicly available, on acceptance.

A. Adding an SSM cache for low-latency serving

In autoregressive inference, including SSM-based inference, each new token is generated based on some **context**, which itself is a function of previously generated tokens. So every time a token is to be generated, the context is reconstructed by processing the prior tokens. As such, a key performance issue in SSM-based inference is the ***repeated reconstruction of context*** for each new token, which inflates end-to-end latency. This issue is further *exacerbated under tensor parallelism (TP)* as the context must be replicated on each GPU, ***bloating its memory usage*** [8].

To address this, we introduce an ***SSM cache*** (see Figure 2) that persists the minimal per-layer context needed to resume generation from the end of the prompt without rescanning

the prefix. With this SSM cache, the inference naturally decomposes into two phases: *prefill*, which is a one-time phase that processes the full input prompt once to populate the cache; and *decode*, which reads from the cache to get context from the previously generated tokens for each new token that needs to be generated. Essentially, at the expense of adding the prefill phase, the SSM cache significantly reduces the redundant processing time of prior tokens in each decode phase.

However, what to store in the SSM cache is not a trivial question under TP. The reusable context in SSMs is *not* a Transformer KV cache; instead it consists of (a) the compact per-layer SSM state produced after processing the prompt, and (b) a short convolution history used by the causal depthwise convolution. Our SSM cache stores exactly these objects, and under TP it is *sharded by channels* (hidden-feature dimensions) so that each GPU stores only the cache entries for the channels it owns. This ensures ***cache reads/writes stay GPU-local*** and prevents the cache itself from adding extra inter-GPU synchronization during decoding to reconstruct the context.

B. Intelligent channel-wise splitting to minimize sync

A central bottleneck in TP is ***inter-GPU communication***: if weight sharding (see Section II-A) is misaligned with how the model packs and consumes activations, the execution repeatedly incurs extra AllReduces just to reassemble intermediate tensors, turning the critical path into a sequence of synchronization points. This is particularly harmful for SSM mixers, where the forward path relies on keeping intermediate layouts contiguous and GPU-local; improper sharding can therefore *force additional layout transforms* which need to be synced, increasing communication volume and reducing effective throughput.

With naive sharding, the number of communication collectives can grow to four per block because packed intermediate tensors must be repeatedly reassembled. Concretely, extra synchronization can be triggered (i) after the *input projection* (to reconstruct the packed activation before it can be chunked), (ii) around the *convolution branch* (when the sharded layout no longer matches the depthwise-convolution channel grouping), (iii) after the *SSM-parameter projection* (to make the per-token parameter vectors consistent across ranks), and (iv) when *merging* the SSM output with the gating branch (to re-form the full residual-stream hidden-state tensor). See Section II-A for a general description of the SSM architecture.

To address this, our TP design employs a **channel splitter** (see Figure 2) that ***shards the weights along channels***, so each GPU owns a disjoint subset of channels and can run the mixer's locality-sensitive operators locally. We apply split-aware channel sharding inside the SSM mixer block: each GPU owns a disjoint subset of channels, and the channel splitter shards the mixer's projection weights so that each rank produces the contiguous channel slices needed by its local downstream computation (depthwise convolution and the SSM update path). Our channel splitter ***lowers the required AllReduces from four (under naive sharding) to two*** per block. First, we perform an AllReduce on the mixer's *SSM-parameter projection* This step ensures every GPU has the *complete* parameter vectors needed for its local gating/SSM computation, without extra reconstruction collectives later. Second, we perform an AllReduce at the *residual-stream boundary* (the end-of-block handoff point where the block's output hidden-state tensor is passed to the next layer). The residual-stream hidden representation is the main activation tensor that flows from block to block (the model's `hidden_states`).

To keep the mixer block's downstream operations local, we ***implement the mixer block's 1D convolution to be channel-separable*** (`Conv1d` on `groups` $=$ `intermediate_size`), i.e., it mixes only across *time* within each channel and never across channels in a single GPU. As a result, when channels are sharded across GPUs, each rank can run the convolution (and the subsequent SSM scan/update) on its local contiguous channel shard without introducing communication in these paths. Because the convolution is applied independently per channel and only slides across time, sharding channels across GPUs preserves correctness without introducing any communication in the convolution step.

C. Handling SSM parameter streams under TP

Beyond projections and convolution, the mixer's SSM update path consumes a set of SSM-related parameters that are produced/packed for efficient execution by the fused SSM kernel. At a high level, these parameters define the per-token state update: Δ controls the input-dependent discretization step (i.e., how the continuous-time dynamics are converted into a per-token update), A governs the recurrent dynamics of the state, B determines how the current token drives the state update, C maps the state back to an output contribution, and D provides a direct residual/skip contribution to the output [1], [5]. However, these parameters do not all behave the same way, so treating them as one uniformly sharded tensor can be both incorrect and inefficient.

In Mamba-style implementations, A and D are learned *per-channel* parameters, while Δ, B, and C are *token-dependent* activation-derived quantities. However, common implementations pack these fields into a single `SSM_parameters` tensor, with fixed column ranges for Δ, B, C, and related fields expected by the fused SSM kernel.

This packing is convenient on a single GPU, but it is not TP-friendly because the packed tensor implicitly assumes a fixed contiguous layout, whereas TP needs a partitioning that depends on the chosen TP degree. If we TP-shard the packed `SSM_parameters` tensor naively, each rank can receive partial slices of multiple logical fields, ***forcing either time-consuming reassembly collectives or extra layout transforms*** before invoking the fused kernel. To avoid this, we ***explicitly unpack the logical fields*** required by the fused SSM kernel and apply a TP-aware placement that keeps the state update GPU-local. Specifically, we *shard* Δ *with channels* (it is token-dependent, the largest stream, and dominates activation-side memory), while ensuring that all remaining quantities required to advance the SSM state for the owned channels are *locally available* on every rank when the fused kernel runs. In our implementation, this means token-dependent terms such as B and C are produced locally from the local activations, while per-channel learned parameters such as A and D are replicated

(or stored) locally as needed. This layout allows each rank to execute the SSM update using the same fused kernel on *contiguous local shards* without any communication inside the SSM path, and it eliminates an otherwise required collective that would be needed to reconstruct packed parameter slices prior to the SSM operation.

D. Quantized AllReduce for the remaining TP synchronizations

After the above design changes, TP communication is confined to a small number of mandatory AllReduces. To further reduce the cost of these remaining yet expensive AllReduce collectives, we optionally apply ***AllReduce quantization***; see Figure 2. Quantizing the AllReduce payload is an intuitive optimization: reducing the communicated precision decreases bytes transferred per collective and therefore the AllReduce latency. Prior work on generic quantization has shown that aggressive low-precision exchange can preserve model quality (often with negligible accuracy loss) while substantially reducing communication overhead [18]–[20].

We implement the AllReduce quantization by only quantizing the communicated tensors to a lower-precision representation (FP16, from FP32) for transfer and reduction, and then dequantizing them back for subsequent computation. With this selective quantization, we target the bottleneck boundary collectives, while leaving the locality-sensitive SSM and convolution kernels unchanged. As shown in our evaluation (Section V-D), our AllReduce quantization reduces bandwidth demand and improves end-to-end throughput.

E. Extending TP to Mamba-2, Falcon-Mamba and Zamba

Our design extends to Mamba-style model families that preserve the same mixer structure. **Falcon-Mamba** uses the same selective SSM mixer, so we reuse our cache layout, channel-wise sharding, and packed-parameter handling unchanged [11]. **Zamba** combines Mamba-style mixers with a lightweight attention component, so we apply our TP design to the mixer layers and use standard column-split TP for attention where applicable [12]. **Mamba-2** follows the same high-level mixer pipeline, allowing us to reuse the same sharding and communication principles while adapting only to its packed-parameter layout [5].

V. Evaluation

We now present our experimental evaluation results highlighting the performance gains afforded by our tensor-parallel (TP) inference design for SSMs. We start in Section V-A by describing our experimental setup, SSM models employed, the baselines for comparison, and the experimental methodology. We then present our key performance evaluation results on the increased prompt lengths enabled by our TP design (Section V-B) and its throughput gains (Section V-C), including with quantization (Section V-D), and an ablation study (Section V-E) to highlight how our TP design contributes to the observed performance improvements.

A. Experimental setup and methodology

We evaluate our TP design on two multi-GPU platforms, representative of modern AI serving infrastructure.

1) **A6000 PCIe workstation.** A single server with an AMD EPYC Milan 7543P CPU (32 cores) and four NVIDIA RTX A6000 GPUs and PCIe 4.0 interconnect.
2) **A100 NVLINK cluster.** A multi-user cluster (queue `a100`) with AMD Milan CPU (96 cores), four NVIDIA A100 GPUs, and NVLINK interconnect.

Scaling TP to larger GPU counts can increase the cost of collective communication [7], [17], since the remaining AllReduce operations must synchronize across more devices and, in multi-node settings, potentially across slower interconnects such as InfiniBand. In large LLM serving deployments, this issue is typically addressed by composing TP with pipeline parallelism (PP) and data parallelism (DP) [17]: TP is applied within a small, high-bandwidth GPU group, while PP and DP scale the deployment across additional devices or nodes.

Models: We evaluate four SSM-based LLMs: **Mamba** [1], **Mamba-2** [5], **Falcon-Mamba** [11], and **Zamba** [12]. We chose these models to span diverse SSM implementations and pure-SSM versus hybrid designs, enabling a representative evaluation of our tensor-parallel design principles. Because **Mamba** and **Mamba-2** use a higher embedding (hidden) dimension than the other models, we evaluate them up to input/output sequence lengths of 256, whereas we evaluate **Falcon-Mamba** and **Zamba** up to 1024. Evaluating contexts beyond the ranges reported here was infeasible on our A6000 and A100 setups as increasing input length rapidly exhausted GPU memory, leading to OOM (Out-Of-Memory) errors.

Comparison baselines: Our TP design shards the model so that multiple GPUs collaboratively execute each request. Our SSM-aware sharding, as described in Section IV, keeps the recurrent update and cached state local while minimizing synchronization overhead in the mixer. For evaluation, we compare our TP design against two inference serving strategies.

- **Default (1×)** runs the model on a single GPU with no model parallelism. This represents the current, de facto practice.
- **Data parallelism (DP)** replicates the full model across GPUs and distributes requests across replicas. DP is a competitive baseline for throughput, especially at higher request concurrency, but it does not reduce the latency (or work done) per request.

Experimental methodology: For both platforms (A6000 and A100), we run 2-GPU and 4-GPU configurations and sweep across prompt (input) lengths, L_{in}, and generation (output) lengths, L_{out}, to cover short-, medium-, and long-context regimes for all models. Unless stated otherwise, all methods (our TP design and the two comparison baselines) use the same model weights, numerical precision, kernels, and decoding strategy. All experiments use the Simple English Wikipedia (SimpleWiki) corpus as the evaluation dataset [23].

B. Results on enabling longer prompt sizes

A key benefit of our TP design is enabling larger prompt sizes (L_{in}, or input sequence length) for a given GPU memory by parallelizing the model across GPUs. To evaluate this benefit, we fix the batch size at 256, and increase the input sequence length in powers of 2 for each method until we encounter an

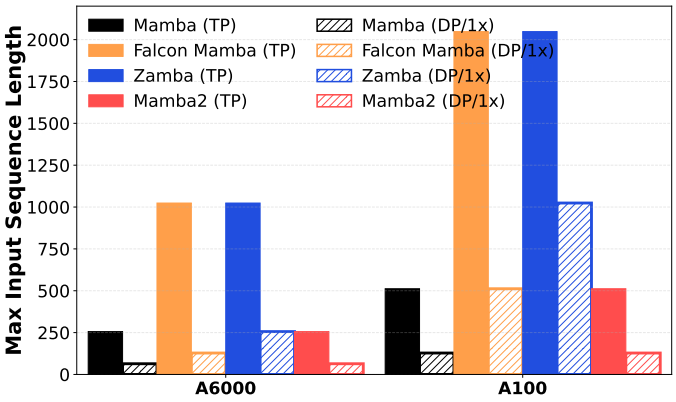


Fig. 3: Maximum input sequence length possible at fixed (256) batch size under our TP design and under DP and 1x.

OOM error. Figure 3 shows maximum successful input length under each method for all models on both GPUs; note that DP and no parallelism ($1\times$) host the entire model on a GPU, and so have the same maximum input length.

We see that our TP design ***consistently supports much longer input prompts*** than DP or 1x on both A6000 and A100. For Mamba and Mamba-2, our TP design supports $4\times$ higher input lengths compared to DP/1x on both GPU architectures (256 vs. 64 on A6000 and 512 vs. 128 on A100). Likewise, for Falcon-Mamba and Zamba, we support 2–$4\times$ higher input lengths. This behavior is expected because our TP design shards model weights and per-layer runtime state across GPUs, reducing the per-device memory footprint. By contrast, DP replicates the full model on every GPU, so each replica hits the same memory ceiling as $1\times$ and cannot admit longer prompts even when more GPUs are available. We note that the memory efficiency enabled by our TP design can also translate to supporting large SSM sizes on a given GPU.

C. Results on throughput gains

We now report the throughput gains afforded by our TP design. For this, we run experiments in a throughput-oriented serving condition. Specifically, for each $\{L_{\text{in}}, L_{\text{out}}\}$ configuration, we *maximize the batch size* separately for each method until the run reaches the memory limit. We do this to ensure a fair comparison of the best achievable throughput under each parallelization strategy. This throughput-maximizing setting is intended to capture deployment-level serving performance rather than isolating compute-only speedup at a fixed batch size. In SSM serving, memory footprint is itself a limiting factor: by sharding model weights, recurrent state, and runtime buffers across GPUs, our TP design can admit larger effective batches before OOM. We thus interpret the reported throughput gains as the combined effect of improved memory efficiency, reduced redundant computation through caching, and parallel execution across GPUs, rather than purely a compute speedup.

Figures 4 and 5 show the throughput improvement afforded by our tensor parallel (TP) design under the A6000 and the A100 GPU clusters, respectively. In each figure, we show the results for each of the four models evaluated (Mamba, Mamba-2 Falcon-Mamba, and Zamba), shown as four columns of subfigures. The first row of subfigures shows the improvement compared to the default no parallelism (referred to as "1x") method and the second row of subfigures shows the improvement compared to data parallelism (DP). In each subfigure, the x-axis is the output sequence length (L_{out}, number of generated tokens). As indicated by the legend, each line in a subfigure corresponds to a fixed input prompt length (L_{in}) and number of GPUs used for parallelism (for DP and TP); we use solid lines to depict 4-GPU and dotted lines to depict 2-GPU results.

1) Performance under NVIDIA A6000: As shown in the first row of Figure 4, our TP design provides significant performance improvement compared to the default, no-parallelism method, with ***throughput gains of 58–98% on 2 GPUs and 226–298% on 4 GPUs*** across the four models shown. The throughput gain achieved by our TP design is primarily because of the scaling benefit it enables of splitting one request across multiple GPUs; while also accommodating larger batch sizes per inference. This is also why the gains are higher for 4 GPUs compared to 2 GPUs. While not significant, the throughput gain for all models slightly increases with longer input/output lengths, indicating that our TP method is most beneficial in the long-context regimes that stress memory pressure.

Compared to DP, our TP design continues to provide substantial performance improvement, with ***throughput gains of 2–59% on 2 GPUs and 9–40% on 4 GPUs*** across the four models as shown in the second row of Figure 4. For Mamba (Figure 4(e)), TP provides consistent gains that grow with the output sequence length and then mildly plateau, across the input-length sweep. For Mamba-2 (Figure 4(f)), trends are similar as observed on Mamba across the input-length sweep. Falcon-Mamba (Figure 4(g)) shows larger gains and a stronger dependence on output sequence length; the benefit is most pronounced at longer contexts where DP cannot reduce per-request work. Zamba (Figure 4(h)) follows a similar pattern but with higher gains and a gradual saturation with output length.

We analyze how TP affects the *prefill* and *decode* phases in Section V-C3.

2) Performance under NVIDIA A100: The first row of Figure 5 reports throughput gains over the default, no-parallelism baseline under the A100 cluster. Similar to the A6000 results, the dominant effect is the splitting of one request across multiple GPUs, so the curves are high and relatively flat across output lengths. Compared to the A6000 results, the performance improvement afforded by our TP design on the A100 cluster is slightly lower, especially for the 4 GPU configuration, with ***throughput gains of 75–96% on 2 GPUs and 150–199% on 4 GPUs*** across the four models. We observe larger TP-over-DP/1x gains on A6000 than on A100, since the A100's higher compute and bandwidth strengthen the DP baseline and leave less headroom for model-parallel speedups. The gains generally grow with longer input/output lengths before saturating, reinforcing that TP is most beneficial in long-context regimes where single-GPU execution becomes limiting.

The second row of Figure 5 shows the throughput gains over DP. For Mamba (Figure 5(e)) and Mamba-2 (Figure 5(f)), TP provides modest but consistent gains that increase with output

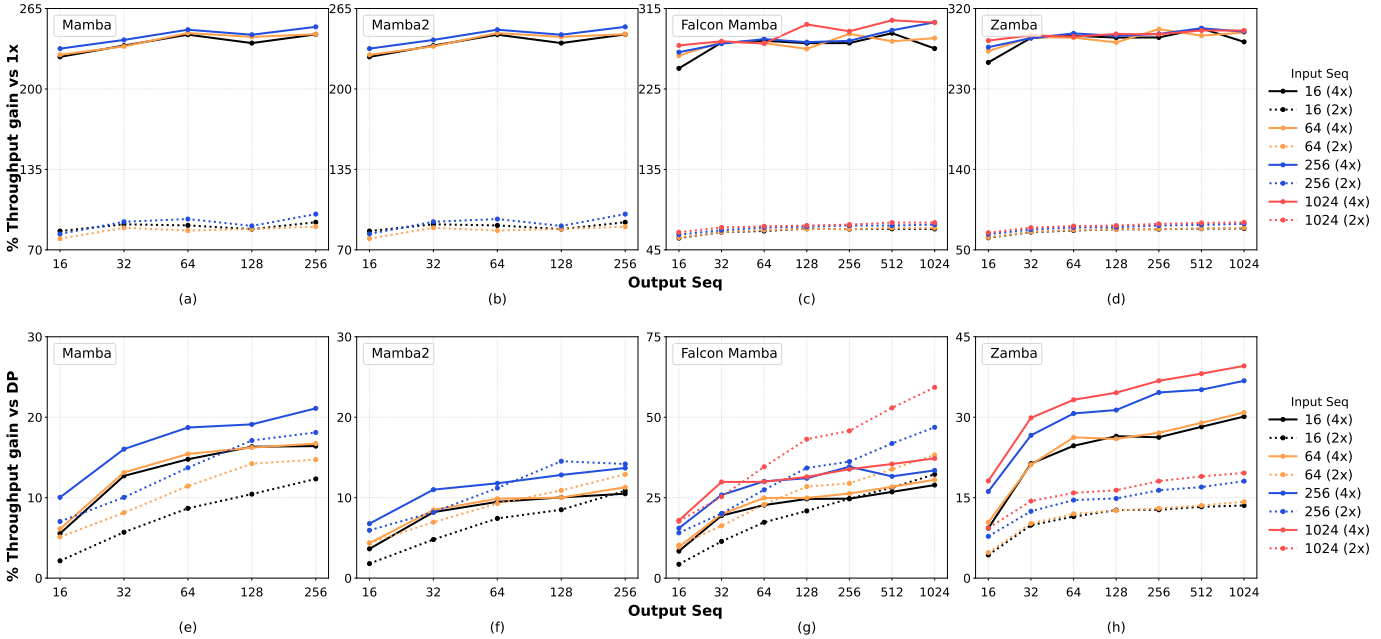


Fig. 4: Throughput gains afforded by our tensor-parallel inference design (for 2-GPU, 4-GPU) for Mamba, Mamba-2, Falcon-Mamba, and Zamba, compared to no parallelism ("1x", first row) and compared to data parallelism ("DP", second row) on the A6000 cluster.

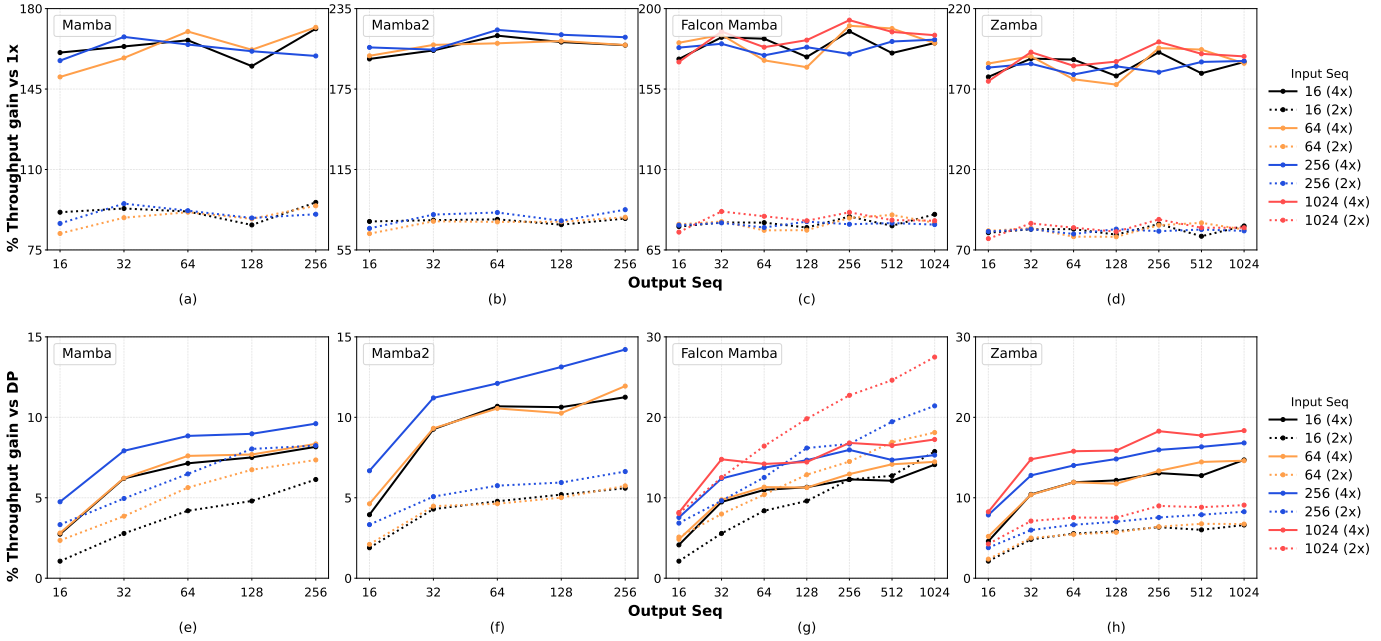


Fig. 5: Throughput gains afforded by our tensor-parallel inference design (for 2-GPU, 4-GPU) for Mamba, Mamba-2, Falcon-Mamba, and Zamba, compared to no parallelism ("1x", first row) and compared to data parallelism ("DP", second row) on the A100 cluster.

sequence length and then plateau, reaching roughly 3–15% on 4 GPUs and 1–8% on 2 GPUs across the input-length sweep. Falcon-Mamba (Figure 5(g)) shows larger gains with stronger dependence on output length, with TP improving throughput by about 4–17% on 4 GPUs and 2–27% on 2 GPUs as output length increases to 1024 tokens. Zamba (Figure 5(h)) follows a similar qualitative trend, achieving 4–16% gains on 4 GPUs and 2–10% gains on 2 GPUs, with gains generally rising with output length. Overall, TP is expected to scale with GPU count (especially over NVLINK and, to a lesser extent, PCIe), but in rare cases such as Falcon-Mamba (Figure 5(g)) and (Figure 4(g)) vs. DP, collective communication can dominate and make 4-GPU TP slower than 2-GPU TP.

3) Understanding the throughput gains: Our TP design impacts the model performance at both the prefill phase (processes the full input prompt once to populate the cache) and the decode phase (reads from the cache to get context from the previously generated tokens for each new token, see Section IV-A). We measure these phases using two standard serving metrics: **time-to-first-token (TTFT)**, dominated by

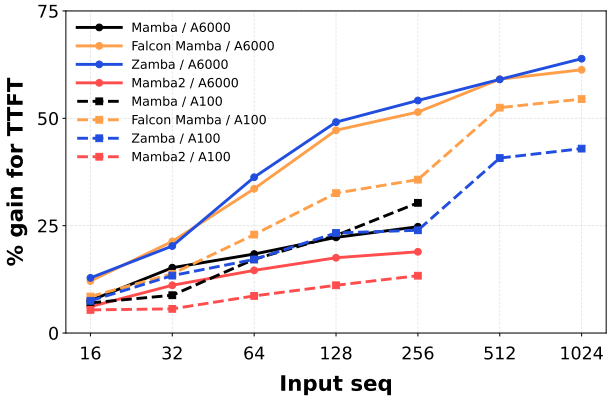


Fig. 6: TTFT gain (vs. DP) afforded by our TP on 4 GPUs.

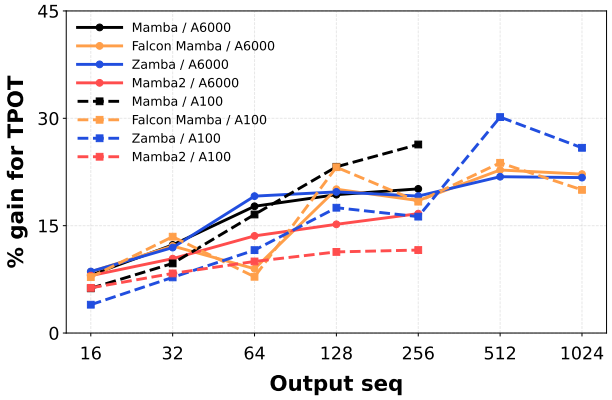


Fig. 7: TPOT gain (vs. DP) afforded by our TP on 4 GPUs.

prefill plus any one-time setup until the first generated token is produced, and **time-per-output-token (TPOT)**, the steady-state per-token cost during decode.

To understand the impact, we now analyze the TTFT and TPOT metrics, that capture the latency of the prefill and decode phases, respectively. We report TTFT and TPOT separately because they scale with different (input or output) sequence dimensions. Note that the throughput results in Figures 4 and 5 show the end-to-end throughput gains, which inherently include the contributions of TTFT and TPOT improvements.

Figure 6 shows the TTFT gains as a function of *input sequence length* for 4-GPU TP relative to DP. Our measurements confirm that TTFT improvements are largely insensitive to the output sequence length, so we only plot TTFT gains for different input lengths. Across all models, TTFT gains increase monotonically with input (prompt) length, with the largest benefits appearing in the long-context regime where prefill dominates. On the A6000 (solid lines), Falcon-Mamba and Zamba scale from roughly 12–13% at 16 tokens to about 61–64% at 1024 tokens, but the gains slow down at longer prompts (e.g., only a small increase from 512 to 1024), indicating a mild plateauing effect as fixed overheads are amortized and the prefill path becomes increasingly bandwidth/compute limited. Mamba shows the same qualitative trend (rising to $\sim$25% by 256 tokens); recall that Mamba encounters OOM error beyond 256 input length, so we only show Mamba results until then. On the A100 (dotted lines), gains are generally smaller at short prompts but remain substantial at long contexts: Falcon-Mamba gains $\sim$54% and Zamba gains $\sim$43% at 1024 tokens, again with diminishing returns toward the longest prompts, while Mamba achieves $\sim$30% gains by 256 tokens.

Figure 7 plots TPOT gains as a function of *output sequence length* for TP relative to the baseline. In contrast to TTFT, TPOT is much more strongly characterized by the output sequence length (decode length) than by the prompt length once the prefill state is established; as such, we only plot TPOT gains for different output lengths to capture steady-state decode behavior and how the per-token gains evolve as generation proceeds. Across all models and both GPUs, we find that TPOT gains rise quickly from short decodes and then plateau as the output length increases: once the system reaches steady-state decoding, the per-token work and the placement of the required TP aggregation collectives are essentially constant, so additional generated tokens do not materially change the average time-per-output-token (TPOT).

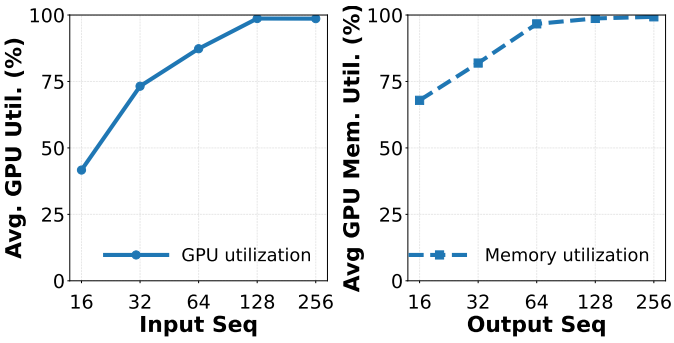
Diminishing returns and utilization. Because TTFT (prefill) and TPOT (decode) exercise fundamentally different bottlenecks (TTFT is primarily compute/SM- and kernel-execution dominated, while TPOT is often dominated by memory footprint and bandwidth), the observed throughput gains under TP can be better understood by profiling the phase-appropriate resource usage (GPU compute utilization for TTFT and memory utilization for TPOT).

Figure 8a shows the average GPU utilization as a function of input length during TTFT (prefill) for Mamba on NVIDIA A6000. We see that the GPU utilization initially rises sharply with increasing input length, but then saturates (at 98.68–98.64% for 128–256 tokens). This indicates that prefill becomes compute-saturated at moderate-to-long prompts, leaving little headroom for TP to extract further throughput gains. Figure 8b shows the average GPU memory utilization as a function of output length during TPOT (steady-state decode). We again see that memory utilization initially increases and then saturates, suggesting decode is increasingly constrained by memory capacity/bandwidth pressure from activations and recurrent/state buffers. TP's remaining benefits are offset by the fixed cost of cross-GPU collectives and synchronization that TP execution entails, similar to the source of diminishing returns in model-parallel language model systems [6], [8].

In summary, while TP increases throughput by enabling larger feasible batch sizes (via parameter/activation sharding) and by distributing the per-token compute across GPUs, its marginal benefit tapers as runs approach fundamental compute/memory limits and as fixed collective overheads become a larger fraction of each step. This behavior is consistent with classic strong-scaling limits: as the non-parallelizable and synchronization components grow in relative cost, speedups saturate (and can even regress) despite adding more parallel resources [24], [25]. This interpretation is consistent with the slight plateauing in Figures 5 and 4 at larger output lengths.

D. Performance enhancement via AllReduce quantization

While our TP design unlocks performance gains via parallel GPU execution, it does introduce additional inter-GPU communication; the results shown thus far include the impact of both these aspects. To further improve the performance afforded



(a) Average GPU utilization (b) Average memory utilization

Fig. 8: GPU and memory utilization during TTFT and TPOT.

by our TP design, we enable quantized AllReduce for tensor-parallel inference and measure its impact on both accuracy and performance; we quantize from the default FP32 precision to FP16. We note that quantization does reduce model accuracy, so this enhancement essentially presents a tradeoff between accuracy and additional throughput gain.

Model	Metric	Value (%)
Mamba	Top-1 (argmax)	98.81
Mamba	Top-5 overlap (Unordered)	99.03
Mamba	Top-5 (Ordered)	89.01

TABLE I: Accuracy impact of quantization for Mamba.

We first quantify the accuracy impact by comparing model outputs against the non-quantized model using agreement-based metrics (Top-1 token match and Top-k overlap). Note that the GPU choice (A6000 or A100) does not impact the model accuracy. Table I shows that our quantized AllReduce incurs only small output perturbations: the next-token prediction under quantization matches the non-quantized version ∼98% of the time and the Top-5 candidate set overlaps ∼99% of the time, suggesting that quantization rarely changes the model's preferred token and typically only perturbs the ranking among close-probability alternatives (reflected in the stricter Top-5 *exact ordering* match of 87–89% across models). These results align with the established view that low-precision communication is a favorable tradeoff when communication becomes a bottleneck [18]–[20].

Figure 9 shows the performance improvement afforded by quantizing AllReduce in our TP design for the Mamba model. Specifically, we plot the throughput gain achieved by our TP design with quantization over TP without quantization. We see that, across all input/output sequence lengths, quantization improves throughput by around 3–12% on A6000 and by around 3–10% on A100, with gains generally increasing with sequence length (larger activations and more collective traffic), peaking at 12.1% for input and output size of 256 on A6000 and 9.6% at the same point on A100. On closer inspection, we find that the (latency) improvement is most pronounced for TTFT where collective synchronization lies on the critical path. On the PCIe-connected A6000, the gains (dotted lines) remain more clearly differentiated across input lengths because AllReduce constitutes a larger fraction of end-to-end latency, whereas on

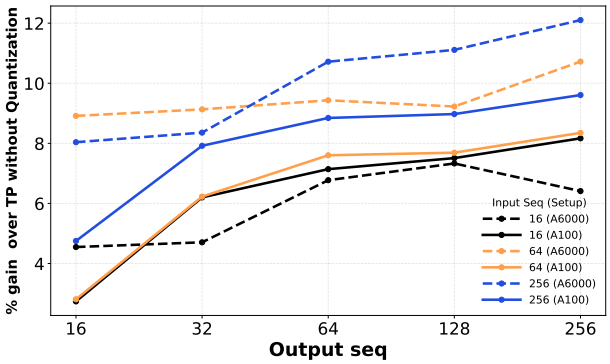


Fig. 9: Throughput gains afforded by quantizing AllReduce in our TP inference design for Mamba as a function of output sequence length.

the NVLINK-connected A100 the gains (solid lines) are more similar as faster collectives reduce the available headroom. This suggests quantized AllReduce becomes increasingly valuable as the communication fabric slows (e.g., PCIe, InfiniBand, multi-node deployments) since TP is often communication-bound and reducing collective payload size directly alleviates the dominant bottleneck. AllReduce quantization improves Falcon-Mamba and Zamba throughput by 12–18%, with average top-1 and top-5 accuracy drops of 2.5% and 1.5%, respectively.

E. Ablation study to examine the benefits of our TP design

To attribute the throughput gains afforded by our TP design (described in Section IV), we now perform an ablation study. Figure 10 shows the latency per token (on log scale) for the Mamba model at 256-token input and output lengths under our TP design; the conclusions are qualitatively similar at other lengths and for other models. The x-axis lists the incremental configurations we designed: only Sharding, Sharding with Caching, and Sharding, Caching, and Quantization. Note that our results in Section V-C employed the Sharding + Caching design and those in Section V-D employed the Sharding + Caching + Quantization design.

We see that both Sharding and Caching provide significant benefits. On the A6000 GPUs, our sharding scheme reduces latency from about 9s to 717.4 ms, a 12.5× improvement; introducing our SSM cache further reduces latency to 83.2 ms, representing an ∼8.6× improvement over just sharding by avoiding redundant prompt re-processing during decode. Adding quantized AllReduce further decreases latency to 73.2 ms (∼12% additional reduction), confirming that bandwidth-optimized synchronization yields measurable benefits even under PCIe interconnects. On the A100 (NVLINK) setup, the same trend holds: sharding alone reduces latency from about 3.1s to 201.3 ms (∼15.6×); caching then reduces latency from 201.3 ms to 18.2 ms (∼11×), and quantized AllReduce brings it down further to 16.2 ms (∼11% reduction). These results reinforce that caching primarily addresses redundant prefill compute, sharding preserves mixer locality and avoids extra synchronization, while quantized AllReduce mitigates communication bottlenecks on the critical path, with stronger relative gains on slower interconnects.

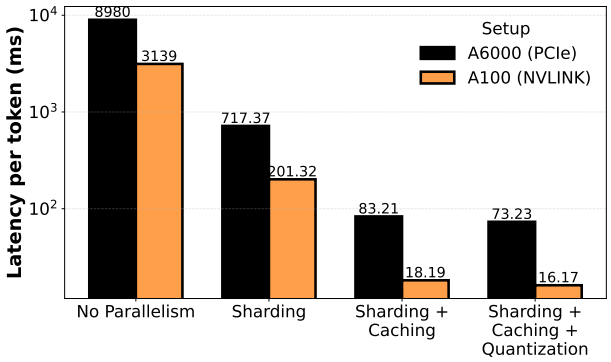


Fig. 10: Ablation results for Mamba (256 input/output length).

VI. Conclusions

This paper presents the first tensor-parallel design for SSM inference and its implementation for four popular SSMs (Mamba, Mamba2, Falcon-Mamba, and Zamba). Our design overcomes the systems challenges of parallelizing SSMs by intelligent channel-wise model splitting to reduce synchronization, introducing an SSM cache to reuse recurrent state, and by implementing AllReduce quantization to further alleviate inter-GPU communication overhead. Our experimental results on two GPU platforms shows that our tensor-parallel SSM inference significantly improves throughput ($\sim$1.4–3.9$\times$) compared to the default, single-GPU inference. Importantly, our design allows us to handle larger SSM configurations, including $\sim$2–4.0$\times$ larger batch sizes and output sequence lengths, due to efficient memory usage under our tensor parallel implementation.

References

[1] A. Gu and T. Dao, "Mamba: Linear-time sequence modeling with selective state spaces," 2024. [Online]. Available: https://arxiv.org/abs/2312.00752

[2] O. Lieber, B. Lenz, H. Bata, G. Cohen, J. Osin, I. Dalmedigos, E. Safahi, S. Meirom, Y. Belinkov, S. Shalev-Shwartz, O. Abend, R. Alon, T. Asida, A. Bergman, R. Glozman, M. Gokhman, A. Manevich, N. Ratner, N. Rozen, E. Shwartz, M. Zusman, and Y. Shoham, "Jamba: A hybrid transformer-mamba language model," 2024. [Online]. Available: https://arxiv.org/abs/2403.19887

[3] R. Xu, S. Yang, Y. Wang, Y. Cai, B. Du, and H. Chen, "Visual mamba: A survey and new outlooks," 2024. [Online]. Available: https://arxiv.org/abs/2404.18861

[4] T. Huang, X. Pei, S. You, F. Wang, C. Qian, and C. Xu, "Localmamba: Visual state space model with windowed selective scan," in *Computer Vision – ECCV 2024 Workshops: Milan, Italy, September 29–October 4, 2024, Proceedings, Part XI*, 2025.

[5] T. Dao and A. Gu, "Transformers are ssms: generalized models and efficient algorithms through structured state space duality," in *Proceedings of the 41st International Conference on Machine Learning*, ser. ICML'24. JMLR.org, 2024.

[6] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," 2020. [Online]. Available: https://arxiv.org/abs/1909.08053

[7] R. Y. Aminabadi, S. Rajbhandari, M. Zhang, A. A. Awan, C. Li, D. Li, E. Zheng, J. Rasley, S. Smith, O. Ruwase, and Y. He, "Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale," 2022. [Online]. Available: https://arxiv.org/abs/2207.00032

[8] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," 2023. [Online]. Available: https://arxiv.org/abs/2309.06180

[9] A. Gu, K. Goel, and C. Ré, "Efficiently modeling long sequences with structured state spaces," 2022. [Online]. Available: https://arxiv.org/abs/2111.00396

[10] AWS Labs, "State space models for aws neuron," 2024, implementing Mamba-2 training on Trainium using neuronx-distributed. [Online]. Available: https://github.com/awslabs/state-space-models-neuron?tab=readme-ov-file

[11] J. Zuo, M. Velikanov, D. E. Rhaiem, I. Chahed, Y. Belkada, G. Kunsch, and H. Hacid, "Falcon mamba: The first competitive attention-free 7b language model," 2024. [Online]. Available: https://arxiv.org/abs/2410.05355

[12] P. Glorioso, Q. Anthony, Y. Tokpanov, J. Whittington, J. Pilault, A. Ibrahim, and B. Millidge, "Zamba: A compact 7b ssm hybrid model," 2024. [Online]. Available: https://arxiv.org/abs/2405.16712

[13] J. Zuo, M. Velikanov, I. Chahed, Y. Belkada, D. E. Rhayem, G. Kunsch, H. Hacid, H. Yous, B. Farhat, I. Khadraoui, M. Farooq, G. Campesan, R. Cojocaru, Y. Djilali, S. Hu, I. Chaabane, P. Khanna, M. E. A. Seddik, N. D. Huynh, P. L. Khac, L. AlQadi, B. Mokeddem, M. Chami, A. Abubaker, M. Lubinets, K. Piskorski, and S. Frikha, "Falcon-h1: A family of hybrid-head language models redefining efficiency and performance," 2025. [Online]. Available: https://arxiv.org/abs/2507.22448

[14] W. Kim, Y. Lee, Y. Kim, J. Hwang, S. Oh, J. Jung, A. Huseynov, W. G. Park, C. H. Park, D. Mahajan, and J. Park, "Pimba: A processing-in-memory acceleration for post-transformer large language model serving," ser. MICRO '25, 2025. [Online]. Available: https://doi.org/10.1145/3725843.3756121

[15] Y. Cheng, L. Wang, Y. Shi, Y. Xia, L. Ma, J. Xue, Y. Wang, Z. Mo, F. Chen, F. Yang, M. Yang, and Z. Yang, "Pipethreader: Software-defined pipelining for efficient DNN execution," in *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI '25)*, 2025. [Online]. Available: https://www.usenix.org/conference/osdi25/presentation/cheng

[16] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, M. Bernauer, B. Catanzaro *et al.*, "Efficient large-scale language model training on GPU clusters using Megatron-LM," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21)*, 2021.

[17] S. Li, H. Liu, Z. Bian, J. Fang, H. Huang, Y. Liu, B. Wang, and Y. You, "Colossal-AI: A unified deep learning system for large-scale parallel training," *arXiv preprint arXiv:2110.14883*, 2021. [Online]. Available: https://arxiv.org/abs/2110.14883

[18] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns," 2014. [Online]. Available: https://www.isca-archive.org/interspeech_2014/seide14_interspeech.html

[19] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "QSGD: Communication-efficient SGD via gradient quantization and encoding," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/6c340f25839e6acdc73414517203f5f0-Abstract.html

[20] T. Vogels, S. P. Karimireddy, and M. Jaggi, "PowerSGD: Practical low-rank gradient compression for distributed optimization," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[21] J. Xin, M. Canini, P. Richtárik, and S. Horváth, "Global-qsgd: Allreduce-compatible quantization for distributed learning with theoretical guarantees," in *Proceedings of the 5th Workshop on Machine Learning and Systems*, 2025, pp. 216–229.

[22] NVIDIA, "Transformer engine release notes – release 2.5," https://docs.nvidia.com/deeplearning/transformer-engine-releases/release-2.5/release-notes/index.html, Jul. 2025, accessed: 2026-01-11.

[23] W. Coster and D. Kauchak, "Simple English Wikipedia: A new text simplification task," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics*.

[24] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[25] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the AFIPS Spring Joint Computer Conference*, 1967, pp. 483–485.