veScale-FSDP: Flexible and High-Performance FSDP at Scale

Zezhou Wang [*1,2] **Youjie Li** [*1] **Zhiqi Lin** [*1] **Jiacheng Yang** [*1] **Cong Xie** [1] **Guanyu Feng** [1] **Zheng Zhong** [1] **Ziyue Huang** [1] **Hongyu Zhu** [1] **Zhi Zhang** [1] **Yanghua Peng** [1] **Xin Liu** [1]

Abstract

Fully Sharded Data Parallel (FSDP), also known as Zero Redundancy Optimizer (ZeRO), is widely used for large-scale model training, because of its memory efficiency and minimal intrusion on model code. However, existing FSDP systems rely on fixed element-wise or row-wise sharding formats that conflict with block-structured computations. As a result, they struggle to support modern structure-aware training methods, including block-wise quantization and non-element-wise optimizers such as Shampoo and Muon. In addition, today's implementations incur communication and memory overheads that degrade efficiency at the scale of tens of thousands of GPUs. We introduce veScale-FSDP, a novel FSDP system that combines `RaggedShard`, a flexible sharding format, with a structure-aware planning algorithm to deliver both flexibility and performance. veScale-FSDP enables zero-copy FSDP communications and natively supports block-wise quantization and non-element-wise optimizers, achieving 5∼66% higher throughput and 16∼30% lower memory usage than existing FSDP systems, while scaling efficiently to tens of thousands of GPUs.

1 Introduction

Large language models (LLMs) have become a transformative technology in everyday applications. Driven by the scaling law (Kaplan et al., 2020), LLMs now reach billions of parameters and approach human-level performance in various domains. Training such giant models requires parallelization techniques that distribute the model and optimizer states across thousands of GPUs (Jiang et al., 2024). Among these, Fully Sharded Data Parallel (FSDP) (Zhao et al., 2023; PyTorch, 2024; Megatron, 2025), also known as DeepSpeed ZeRO (Rajbhandari et al., 2020), is one of the most fundamental techniques. FSDP is often the first choice because of its memory efficiency and flexible data-parallel programming paradigm that is decoupled from model architecture. When additional scaling is needed, FSDP can be combined with other forms of parallelism (Smith et al., 2022; Ma et al., 2025).

However, existing FSDP systems struggle to support modern structure-aware training methods. State-of-the-art models use non-element-wise optimizers such as Shampoo (Gupta et al., 2018) and Muon (Jordan et al., 2024), and block-wise quantized training as used in DeepSeek-V3 (Liu et al., 2024), all of which require tensor blocks to remain intact under sharding. The core limitation is that existing FSDP frameworks shard model states either element-wise (Rajbhandari et al., 2020; Zhao et al., 2023) or row-wise (PyTorch, 2024; Megatron, 2025), producing sharding boundaries that often misalign with the required block structure. Consequently, either model developers must intrusively modify the model or optimizer code to match tensor boundaries, or system developers must handle complex boundary checks, padding, and additional communication logic.

Beyond inflexibility, existing FSDP systems fall short of production throughput and memory targets, where we aim to extract every bit of hardware efficiency. GPU Memory is the tighter constraint: in shared clusters, jobs run out of memory or will operate at the memory limit incurring expensive device-side frees, prompting over-provisioning that leaves GPU resources wasted. These demands become even more critical when scaling training to over 10K GPUs and trillions of parameters. Few existing FSDP systems can scale to this level while maintaining efficiency. DeepSpeed ZeRO (Rajbhandari et al., 2020) pioneered FSDP research but suffers from fragmented AllGather operations (Halilakin, 2024) and inefficient memory management (Xu, 2024). PyTorch FSDP1 (Zhao et al., 2023) addresses some AllGather inefficiency, but incurs slow ReduceScatter (Zhao et al., 2025) and does not solve memory overhead (Xu, 2024). PyTorch FSDP2 (PyTorch, 2024) improves memory management (Gu et al., 2023) but introduces a high tensor copy overhead. Meanwhile, both FSDP1 and FSDP2 suffer from slow collectives due to unaligned communication buffers (Wu et al., 2025; NCCL, 2025a). Megatron-FSDP (Megatron,

[*]Equal contribution [1]ByteDance Seed [2]University of Washington; Work done during internship at ByteDance Seed. Correspondence to: Youjie Li and Yanghua Peng <youjie.li@bytedance.com and pengyanghua.yanghua@bytedance.com>.

Proceedings of the 9^{th} MLSys Conference (Industry Track), Bellevue, WA, USA, 2026.

2025) adopts a zero-copy design that avoids FSDP2's tensor copy overhead but requires extra padding, increasing both communication and memory costs.

To address these flexibility and performance shortcomings, we present veScale-FSDP, a novel FSDP system that preserves the PyTorch-native `fully_shard` API while rearchitecting the PyTorch FSDP2 backend for flexible and high-performance training at scale:

▷ For flexibility, veScale-FSDP introduces a novel sharding format, `RaggedShard`, which supports arbitrary sharding granularity with custom block sizes for structure-aware training, while seamlessly composing with existing PyTorch DTensor sharding formats.

▷ For performance, veScale-FSDP introduces a planning algorithm that rearranges `RaggedShard` tensors to maximize communication efficiency while respecting their desired sharding blocks. We formulate planning as an NP-hard optimization problem and use practical polynomial-time heuristics that achieve high-quality solutions in practice.

▷ veScale-FSDP further provides a high-performance primitive, Distributed Buffer (`DBuffer`), that backs `RaggedShard` tensors with slices of a global buffer, enabling zero-copy access, reducing communication overhead, and reducing memory fragmentation via batched memory allocations.

Our extensive evaluations demonstrate that veScale-FSDP outperforms all existing FSDP systems on both dense and sparse LLMs across different scales, achieving 5∼66% higher throughput and 16∼30% lower memory usage while scaling efficiently to tens of thousands of GPUs. In addition, case studies show that veScale-FSDP natively accommodates both non-element-wise optimizers such as Muon (Jordan et al., 2024) and block-wise quantization methods such as 8-bit Adam (Dettmers et al., 2022). veScale-FSDP has been deployed in production for most training workloads at ByteDance Seed, and is portable without relying on internal infrastructure.

`RaggedShard` code is open-sourced at `https://github.com/volcengine/veScale`.

2 Background and Motivation

2.1 Structure–Aware Training

Structure-aware training underpins frontier models such as Kimi K2 (Team et al., 2025), Gemini (Team et al., 2024), and DeepSeek-V3 (Liu et al., 2024), and is becoming increasingly important. Representative examples include:

Matrix Optimizers. Matrix-based optimizers such as Shampoo (Gupta et al., 2018) and Muon (Jordan et al., 2024) can converge faster than AdamW (Wen et al., 2025). Their updates operate on parameters in their original 2D matrix rather than on independently sharded elements. In practice, this requires gathering each matrix on a selected device before performing the matrix-level update, and then redistributing the update back.

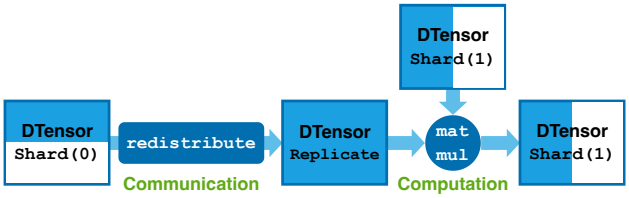


Figure 1. Distributed Tensor (DTensor) for flexible communication and computation. The figure shows an example of DTensors executing a sharded matrix multiplication (`matmul`) on a device. A row-sharded (`Shard(0)`) DTensor is first redistributed to a replicated (`Replicate`) DTensor so that it can be multiplied with a column-sharded (`Shard(1)`) DTensor, producing a column-sharded (`Shard(1)`) DTensor. The blue part in each DTensor indicates the materialized local tensor on that device.

Block-wise Quantization. Quantizing model weights (Liu et al., 2024) and optimizer states (Dettmers et al., 2022) is widely used to improve training efficiency. Block-wise quantization is a common technique because per-block scaling factors mitigate the accuracy loss from reduced precision while preserving system efficiency. However, communication-free block-wise quantization requires each quantization block to reside entirely on a single device. If parameters are sharded without respecting block boundaries, devices must exchange scaling-factor metadata, complicating the implementation and reducing the efficiency gains.

2.2 DTensor and JaggedTensor

Distributed Tensor (DTensor) (Xu et al., 2021; The PyTorch Team, 2024; Li et al., 2025) is a promising primitive of PyTorch that provides an opportunity towards structure–aware training. It represents a logical global tensor distributed across multiple devices, with each device holding its local tensor. DTensor supports three sharding formats (placements): `Shard(dim)`, which evenly shards a global tensor along a tensor dimension; `Replicate`, which replicates the global tensor on every device; and `Partial`, in which each device holds a partial value of the global tensor that must be reduced across devices to materialize the full result. It also provides a `redistribute` API that converts a DTensor between placements with implicit collective communications. Additionally, DTensors can be computed directly from operators such as `matmul`, as shown in Figure 1. However, a fundamental limitation remains for structure-aware training: the `Shard` format cannot represent the block-wise sharding needed for quantization or the

Table 1. Interleaved copy overhead in FSDP2 for GPT-OSS-120B on 64 H800 GPUs. The table reports the time of interleaved Copy-Out relative to AllGather on the AllGather path, and the time of interleaved Copy-In relative to ReduceScatter on the ReduceScatter path. `Shard(0)` is the default sharding format for each parameter, while `Shard(1)` is used when `Shard(0)` incurs large padding.

	AllGather path		ReduceScatter path	
	AllGather	Copy-Out	ReduceScatter	Copy-In
Shard(0)	43.71 ms	5.22 ms	94.24 ms	12.37 ms
Shard(1)	44.35 ms	13.72 ms	95.36 ms	23.14 ms

uneven sharding required by matrix optimizers.

JaggedTensor/NestedTensor on a single device (PyTorch, 2025a;b; TensorFlow, 2025) is a PyTorch/TensorFlow primitive for representing tensors whose last dimension is jagged. For example, a 2D tensor whose rows may have different lengths. These primitives cannot express sharding with block-level granularity, but they offer a useful hint for how veScale-FSDP can support structure-aware sharding in distributed training.

2.3 ZeRO and FSDPs

DeepSpeed ZeRO (Rajbhandari et al., 2020) pioneered this line of FSDP research. Its core idea is to concatenate a layer of tensors (parameters, gradients, and optimizer states) and then shard each concatenated tensor across devices, where tensors may be irregularly partitioned across device boundaries. ZeRO only unshards a layer using AllGather before the forward and backward passes, and reduces the layer gradients using ReduceScatter back across devices. Such a sharding design is fundamentally element-wise and cannot support structure-aware training.

FullyShardedDataParallel (FSDP1) (Zhao et al., 2023) is the first PyTorch-native ZeRO, following the same sharding format and limitation, but it is optimized for performance.

`fully_shard` (FSDP2) is the second PyTorch-native ZeRO and represents the state of the art in the FSDP community. It replaces the concatenated-shard design with per-parameter sharding, representing each tensor as a `Shard(0)` DTensor. This exposes DTensor flexibility for FSDP parameters in communication, computation, and checkpointing. However, this evenly sharded format still falls short of enabling structure-aware training. Moreover, FSDP2 introduces performance overheads by copying parameters into and from interleaved memory addresses, as shown in Figure 2 and Table 1.

Megatron-FSDP (Megatron, 2025) is the most recent FSDP prototype optimized for speed. It forgoes FSDP2's design and reverts to FSDP1's concatenated sharding to avoid copying overhead, while further optimizing performance. However, Megatron-FSDP introduces a special mechanism to make a concatenation-sharded tensor appear as a `Shard(0)` DTensor so that model checkpointing can reuse DTensor. This mechanism inserts padding into the concatenation so that tensors are sharded row-wise along device boundaries rather than element-wise. Without careful padding planning, the concatenation size can grow significantly, increasing both memory usage and communication volume. Moreover, row-wise sharding still falls short of supporting structure-aware training. Its granularity is fixed by the parameter layout and may not match what the block-wise quantization requires.

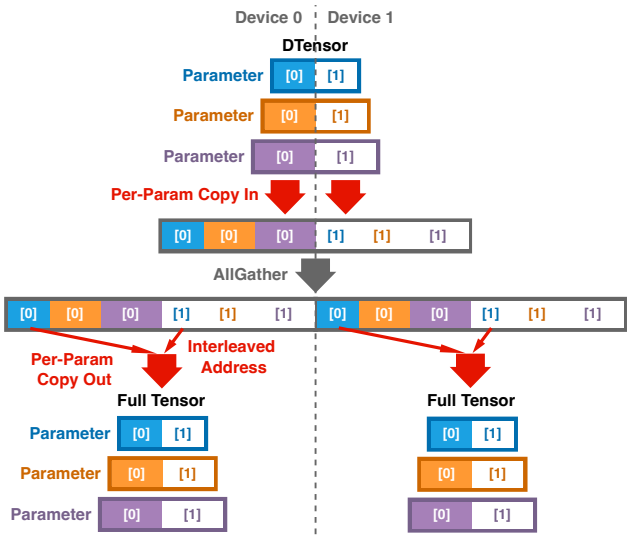


Figure 2. The copy overhead in the FSDP2 design. Each parameter is represented as a `Shard(0)` DTensor and is evenly sharded across devices. Before AllGather, the input communication buffer on each device needs to copy in the local shards of all parameters. After AllGather, the output communication buffer stores each parameter in interleaved memory addresses rather than a contiguous address. As a result, FSDP2 must copy out each parameter from the output buffer to materialize the full tensor with a contiguous address needed for computation. The ReduceScatter path is a reversed one and incurs interleaved Copy-In overhead.

3 Overview

To address both flexibility and performance challenges, we present veScale-FSDP, a novel FSDP system. Figure 3 provides an overview. veScale-FSDP allows model developers to build sophisticated large models (e.g., with sparse MoE structures) and structure-aware optimizers (e.g., with non-element-wise operators) that achieve better model quality. At the same time, developers can parallelize these models and optimizers through PyTorch's native `fully_shard` API, as in FSDP2, without intrusive modifications to model

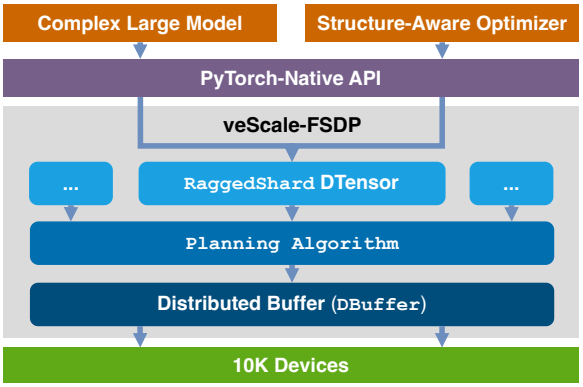


Figure 3. veScale-FSDP overview.

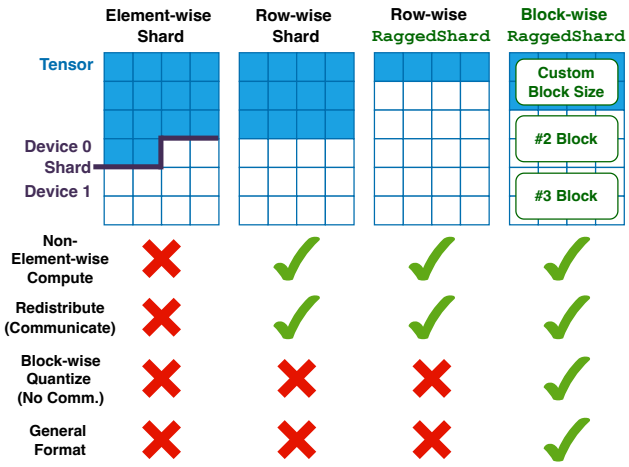


Figure 4. Flexibility comparison of different sharding formats.

or optimizer code. During parallelization, complex model and optimizer operators can still preserve single-device semantics through a new sharding format, `RaggedShard`, which supports arbitrary sharding granularities over contiguous storage and arbitrary distributions across devices for each DTensor (§4). Under the hood, `RaggedShard` DTensors are grouped for bucketed communication. To optimize performance, their layouts are rearranged by a planning algorithm derived from an NP-hard optimization problem. The planned layouts are then mapped to a Distributed Buffer (`DBuffer`), a new primitive that enables zero-copy communication path with minimal overhead (§5). Together, these components enable veScale-FSDP to scale efficiently to 10K GPUs in real production deployments.

4 RaggedShard for Flexibility

This section proposes `RaggedShard`, a novel and general sharding format to increase FSDP's flexibility for complex models and structure-aware optimizers.

Existing sharding formats. Figure 4 summarizes existing sharding formats for FSDP tensors. The most common and conventional format is the *Element-wise Shard*, where tensors are partitioned arbitrarily across devices without respecting structural boundaries. This leads to dangling elements on each device and a loss of tensor shape and stride information. Consequently, this format severely limits flexibility: (i) non-element-wise operations such as matrix multiplications break on misaligned elements, (ii) collective communication for redistributing tensors across dimensions becomes complex and inefficient, and (iii) quantization methods such as block-wise FP8 quantization cannot align block sizes with random shard boundaries, introducing substantial communication overhead for cross-boundary synchronization. Unfortunately, this element-wise sharding design remains the backbone of DeepSpeed and FSDP1.

The second format is the *Row-wise (Even) Shard*, where a tensor is evenly partitioned along a dimension, with equal-sized shards assigned to each device. This design improves flexibility by enabling non-element-wise computations on sharded tensors and allowing dimension redistribution via `All2All` collectives. However, it still faces challenges with block-wise quantization, as evenly divided shards are not guaranteed to align with block boundaries. This row-wise sharding format serves as the foundation of FSDP2.

The `RaggedShard` format. Inspired by JaggedTensor/NestedTensor (PyTorch, 2025a;b; TensorFlow, 2025), we propose the `RaggedShard` format for DTensor. `RaggedShard` increases flexibility by supporting arbitrary ***sharding granularity*** in contiguous memory, i.e., the size of the atomic non-shardable block, and arbitrary ***sharding distribution***, i.e., the number of such blocks placed on each device. The sharding granularity can be defined over contiguous tensor units, such as elements, rows, or higher-dimensional planes. Figure 4 shows a simple example: when the sharding granularity is set to one tensor row, `RaggedShard` yields *Row-wise* `RaggedShard`, where different devices may hold different numbers of rows. A similar idea has been prototyped in the model checkpointing mechanism of Megatron-FSDP.

The most flexible sharding format is the *Block-wise* `RaggedShard`, where the sharding granularity is defined as a tensor block with a customizable shape. For example, a tensor may be partitioned into three 2D blocks, with one block placed on device 0 and two blocks on device 1 (see Figure 4). This format not only supports non-element-wise computation and efficient redistribution but also enables block-wise quantization with perfect alignment between quantization blocks and shard boundaries. In fact, the block-wise `RaggedShard` generalizes all previous sharding formats through different choices of block size.

Composing with existing sharding formats. DTensor has

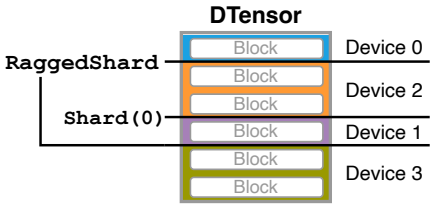


Figure 5. Composability of `RaggedShard` with existing evenly sharded DTensor placements in 2D parallelism schemes such as FSDP×EP (Expert Parallelism).

been widely used to express tensor partitions in parallelization strategies such as Tensor Parallelism (TP) (Shoeybi et al., 2019) and Expert Parallelism (EP) (Lepikhin et al., 2020). It allows tensors to be represented using replicated, partial-value, or evenly sharded placements along a selected dimension. `RaggedShard` extends this capability as an additional DTensor placement.

To support combinations of multiple parallelization strategies, `RaggedShard` needs to compose cleanly with existing DTensor placements. `RaggedShard` is orthogonal to both replicated and partial-value placements and veScale-FSDP specially handles the `Shard` placement. In practice, TP uses `Shard(0)` and `Shard(1)` for column- and row-wise Tensor Parallelism; EP can be encoded as `Shard(0)` along the expert dimension. By convention, EP/TP is applied before FSDP. In PyTorch, however, the DTensor placement list is organized in the opposite order of conceptual application (see Figure 5): a tensor shown with placements (`RaggedShard`, `Shard(0)`) is partitioned as `Shard(0)` followed by `RaggedShard`. veScale-FSDP reconciles this by: (i) for `Shard(0)`, introducing a dedicated placement `StridedRaggedShard` that carries reordering/stride metadata and performs the required reshuffle when materializing the full tensor; and (ii) for `Shard(dim>0)`, adapting the ragged sharding granularity so it never cuts into that dimension by choosing the granularity as Least Common Multiple (LCM) of the tensor stride of that dimension and user-defined granularity.

Meanwhile, `RaggedShard`, as an extended DTensor placement, offers checkpointing capability by directly reusing DTensor-based checkpointing stacks (e.g., PyTorch Distributed Checkpoint (PyTorch Team, 2025)) for failure recovery and also inheriting their optimizations such as communication-free sharded checkpointing.

5 Grouped RaggedShard for Performance

This section discusses how to group `RaggedShard` DTensors for efficient communication. We formulate the underlying optimization problem, prove the NP-hardness of the problem, and present a polynomial-time heuristic algorithm that achieves high-quality solutions in practice.

Challenges for efficient communication. As is well known in the systems community, collective communication relies on tensor bucketing or grouping to maximize network utilization (Li et al., 2020; Zhao et al., 2023). The same principle applies to communication over `RaggedShard` DTensors in veScale-FSDP. However, efficiently grouping `RaggedShard` tensors is non-trivial and straightforward approaches can lead to significant inefficiencies. Figure 6(a) illustrates three major sources of inefficiency:

- *Sharded block*: Tensors are concatenated into a communication buffer without respecting block boundaries, causing individual blocks to be split across devices; this violates the abstraction of Block-wise `RaggedShard` and incurs additional communication for quantization.
- *Non-contiguous tensor memory*: Padding may be inserted at the ends of communication buffers to satisfy collective alignment requirements (Wu et al., 2025; NCCL, 2025a) or equal-size constraints across devices (NCCL, 2025b), but may fall within a tensor. This breaks tensor contiguity and introduces interleaved copy overhead, similar to the Copy-Out after AllGather in Figure 2.
- *Imbalanced load*: Differences in tensor sizes, block sizes, or padding sizes may lead to unequal buffer sizes across devices. This breaks the symmetry of collective communication and results in underutilized network bandwidth.

Towards efficient communication. To efficiently group `RaggedShard` DTensors, we propose a two-step approach that addresses the above challenges: first permute tensors, and then pad between them rather than padding within individual tensors, as illustrated in Figure 6(b). The key idea is to balance tensor and block sizes across devices while aligning block boundaries in the sharded communication buffer so that blocks are placed contiguously. This approach inevitably introduces some padding overhead, which must be carefully minimized to reduce both memory usage and communication volume.

Optimization problem formulation. Formally, the proposed approach can be formulated as an optimization problem. Let $\mathcal{T} = \{t_1, t_2, \ldots, t_n\}$ denote a set of `RaggedShard` DTensors, which are sharded across m devices. Each DTensor $t \in \mathcal{T}$ has a block size of g_t, a total tensor size (in elements) e_t, and therefore contains $u_t = e_t/g_t$ sharding blocks. We allocate a global communication buffer and place each t in the buffer as a contiguous memory interval $[\ell_t, r_t)$. The decision variables are a uniform per-device buffer size S and the interval endpoints $\{\ell_t, r_t\}_{t\in\mathcal{T}}$. The global communication buffer therefore has total size mS, partitioned into m equal-sized device-local buffers, where device k owns a memory interval of $[(k-1)S, kS)$ for

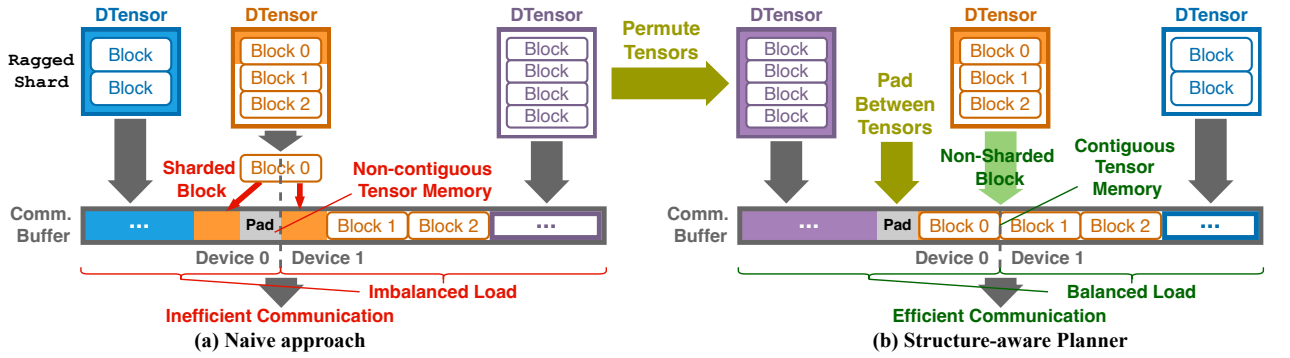


Figure 6. Communication layouts for grouped `RaggedShard` DTensors. (a) shows a naive strategy that concatenates tensors directly into the communication buffer. This can split blocks across shard boundaries, introduce padding within tensors, and produce unequal per-device buffer sizes, leading to inefficient communication. (b) shows our structure-aware planner that carefully plans the communication layout – first permutes tensors and then inserts padding between tensors rather than within them. This preserves non-sharded blocks, keeps each tensor contiguous in memory, and balances per-device communication buffers, leading to efficient communication.

$k = 1, \ldots, m$. Our goal is to minimize S subject to three constraints: Non-Sharded Block, Contiguous Tensor Memory, and Balanced Load (Figure 6(b)):

$$\begin{aligned} &\min_{S,\{\ell_t,r_t\}_{t\in\mathcal{T}}} \quad S \\ \text{s.t.} \quad & r_t - \ell_t = e_t \ \wedge \ r_t \le mS, \forall t \in \mathcal{T}, \\ & r_t \le \ell_{t'} \ \vee \ r_{t'} \le \ell_t, \forall t \ne t' \in \mathcal{T}, \\ & kS \le \ell_t \ \vee \ kS \ge r_t \ \vee \ (kS - \ell_t) \equiv 0 \pmod{g_t}, \\ & \qquad \forall t \in \mathcal{T}, \ \forall k = 1, \ldots, m \end{aligned}$$

This optimization problem is ***NP-hard***, as it can be reduced from the classic Partition problem (Garey & Johnson, 1975). Although it can be formulated as an Integer Linear Programming (ILP) problem and solved using off-the-shelf solvers, such methods are impractical at scale. In practice, ILP solvers often take tens of minutes to generate a plan and may even trigger system timeouts. Given that user-defined FSDP wrapping can yield hundreds of parameter groups with diverse sharding block sizes, and deployments may span up to hundreds of thousands of devices, we instead design a polynomial-time heuristic algorithm that achieves near-optimal efficiency in practice.

Heuristic-guided solution. veScale-FSDP introduces a polynomial-time dynamic-programming (DP) buffer-layout algorithm, guided by permutation heuristics that exploit the regularity of transformer models. The optimization difficulty arises from tensor ordering: in principle, any permutation of tensors could be mapped into the global communication buffer, and finding the global optimum would require exploring all permutations. Fortunately, in practice, transformer parameters are highly structured: linear weights dominate the total parameter count, and sharding blocks are often consistent across layers. To leverage this regularity, we consider three tensor orders: (i) the default tensor order; (ii) sorting by sharding block size; and (iii) sorting by tensor shape. Our statistics show that these orders yield optimal or near-optimal results, so we adopt the default order for simplicity and ease of debugging. For non-transformer architectures, alternative tensor orderings can be plugged in without changing the DP algorithm itself.

Given the tensor order, the proposed algorithm applies a DP procedure to place tensors into the smallest possible global buffer while enforcing the aforementioned three constraints. It has a time complexity of $O(|\mathcal{T}|^2 m \log(E) \log(|\mathcal{T}|m))$. Algorithm 1 presents the details.

The core idea is a case analysis of how each tensor aligns with shard boundaries in any valid layout: (1) it lies entirely within a single local shard; (2) it straddles two adjacent shards but does not fully contain a shard; and (3) it fully contains at least one shard. If every tensor falls into cases (1) or (2), feasibility is monotonic in the shard size S: whenever a layout exists for S, it also exists for $S + \Delta$, where Δ is the smallest alignment unit required by the layout. Because every shard includes an inter-tensor boundary, the additional Δ can always be absorbed as padding. If any tensor falls into case (3), the feasible shard sizes must be multiples of $L = \mathrm{LCM}\{\, g_t \mid t \text{ is in case (3)} \,\}$. In this regime, feasibility is monotonic over multiples of L: if kL is feasible, then $(k+1)L$ is also feasible. We therefore binary-search for the minimal feasible S over the corresponding multiples, as shown in Lines 21–25. To avoid enumerating an exponential number of candidate case-(3) sets, we sort tensors by element count and consider only prefixes of this sorted order, yielding a 2-approximation. Within CHECKVALIDSHARD, we define $dp(t, i)$ as the minimum number of devices (shards) required to store all previous tensors and the first i sharding blocks of tensor t. Although the DP state

Algorithm 1 Structure-aware planning for grouped communication of `RaggedShard` DTensors.

Input: ordered tensor list $\mathcal{T}$; per-tensor sharding block size g_t (collectively $G = \{g_t\}$); per-tensor size e_t; per-tensor number of blocks u_t; number of devices m; collective preferred unit size g_{coll}.
Output: minimal uniform per-device buffer size $S^\star$.
Notation: for a candidate shard size S, $dp(t, i; S)$ denotes the minimum number of device-local shards required to place all tensors before t and the first i sharding blocks of tensor t.

function CheckValidShard(S)
  initialize segment records for $dp(\cdot, \cdot; S)$
  for all $t \in \mathcal{T}$ **do**
    $l \leftarrow 0$
    while $l < u_t$ **do**
      // $dp(t, i; S)$ is monotonic in i, so skip intermediate states within a constant segment
      $r \leftarrow \max\{i \in [l, u_t) : dp(t, i; S) = dp(t, l; S)\}$
      record segment $[l, r]$ with value $dp(t, l; S)$
      $l \leftarrow r + 1$
    end while
  end for
  return $dp(t_{\text{last}}, u_{t_{\text{last}}}; S) \leq m$
end function

$g \leftarrow g_{\text{coll}}$
$S^\star \leftarrow +\infty$
for all $g' \in$ SortAscending(G) **do**
  $g \leftarrow$ LeastCommonMultiple(g, g')
  $S' \leftarrow \min\{k * g : k \in N \wedge$ CheckValidShard($k * g$)$\}$
  $S^\star \leftarrow \min(S^\star, S')$
end for
return $S^\star$

space indexes every block position within a tensor, $dp(t, i)$ is monotonic: $dp(t, i) \leq dp(t, i + 1)$. Therefore, each tensor has at most m distinct DP values. In Lines 10–13, we exploit this property by grouping contiguous indices into segments and skipping intermediate $dp(t, *)$ calculations, which yields the stated time complexity.

Distributed Buffer (`DBuffer`) Beyond grouping `RaggedShard` DTensors, the underlying communication buffer also plays a vital role in achieving high communication, computation, and memory efficiency. veScale-FSDP introduces a new primitive, Distributed Buffer (`DBuffer`), for efficient grouped DTensors. Figure 7 shows the design. First, inspired by DTensor, `DBuffer` provides global buffer semantics over an N-dimensional device topology, with a sharding specification along each dimension, abstracting away the complexity of N-D communication and

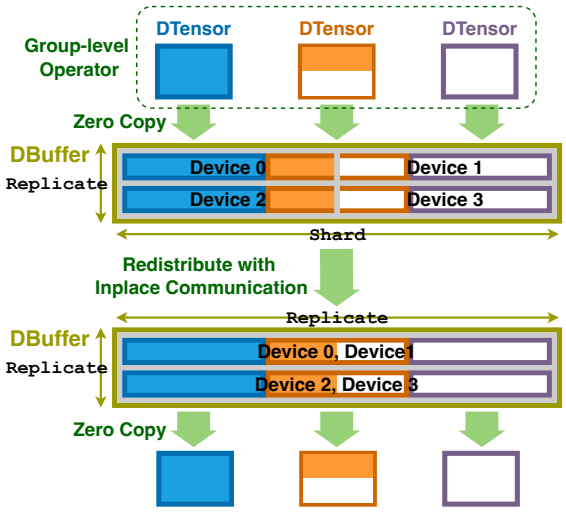


Figure 7. Distributed Buffer (`DBuffer`) for high-performance communication. A 2D `DBuffer` for parameter AllGather is shown; conversely, a 2D `DBuffer` redistributing from `(Partial, Partial)` to `(Replicate, Shard)` implements 2D gradient reduction with ReduceScatter and AllReduce.

operations. Second, `DBuffer` takes a group of tensors and executes group-level operators rather than per-tensor operators. For example, before communication, each tensor may need to launch its own CUDA kernels for add, scale, zero, or copy (which may differ across tensors), incurring fragmented compute overhead and blocking communication. With `DBuffer`, identical kernels across tensors are fused before communication, reducing blocking time. Third, `DBuffer` offers zero-copy access before and after communication by leveraging `RaggedShard`'s planning algorithm and providing a persistent address mapping to each tensor's data pointer, thereby minimizing memory footprint and fragmentation. Finally, `DBuffer` uses in-place communication and computation.

6 Evaluation

Our evaluation answers the following questions:

- How much does veScale-FSDP improve end-to-end training performance over all baseline systems (§6.1)?
- How well does veScale-FSDP scale to large device counts (§6.2), in terms of weak scaling, strong scaling, and model size scaling?
- How are 8-bit Adam and Muon optimizer enabled by veScale-FSDP's customizable sharding granularity and `RaggedShard` DTensor, in both performance and development velocity (§6.3)?
- How does the veScale-FSDP planner minimize padding, and what is the algorithm overhead (§6.4)?
- How much does each component of veScale-FSDP contribute to the training performance (§6.5)?

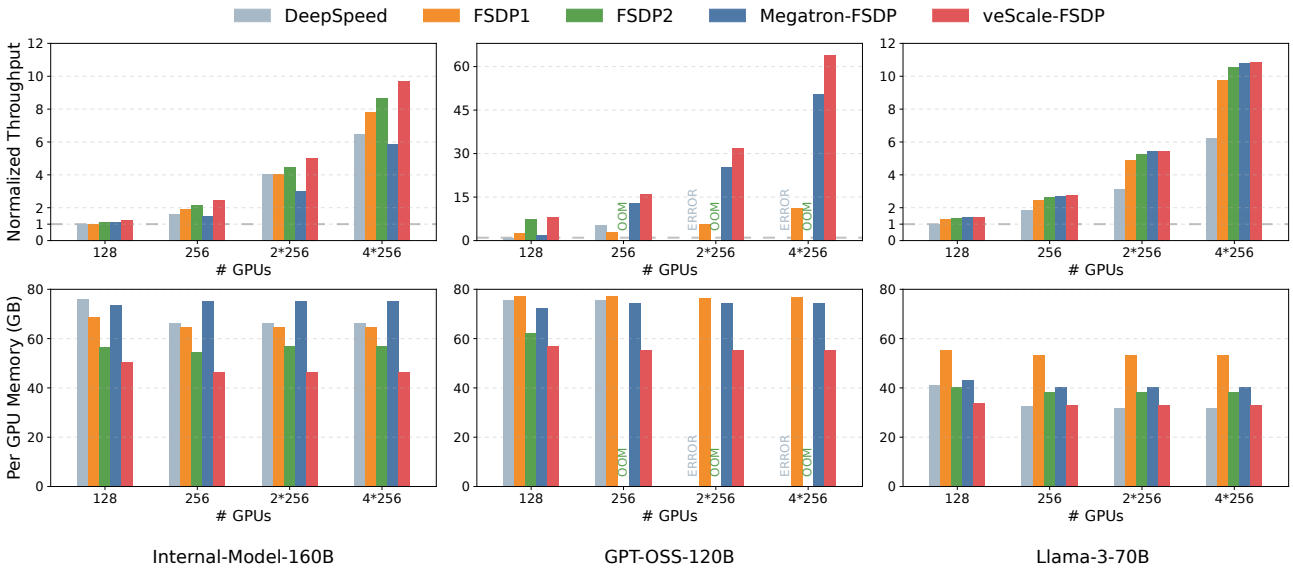


Figure 8. FSDP training performance. Top row: normalized aggregate throughput (tokens/s). Bottom row: peak per-GPU memory (GB). We sweep FSDP (ZeRO-3) at 128/256 GPUs and HSDP with 2- and 4-way replication (2*256, 4*256 GPUs).

Hardware: We ran the experiments §6.1, §6.4, and §6.5 on a NVIDIA H800 cluster; each node contains 8×H800 GPUs (979 BF16 TFLOPS, 80 GB HBM) and 400 GB/s NVLinks. The experiments §6.2 and §6.3 were conducted on a NVIDIA Hopper cluster.

Implementation: veScale-FSDP is implemented with 7.6 K lines of code (LoC) in Python, transparently replacing the backend of FSDP2 while using the same PyTorch-native `fully_shard` API. veScale-FSDP serves as a plug-and-play Python module, compatible with standard PyTorch distributed runtimes and a wide range of PyTorch versions.

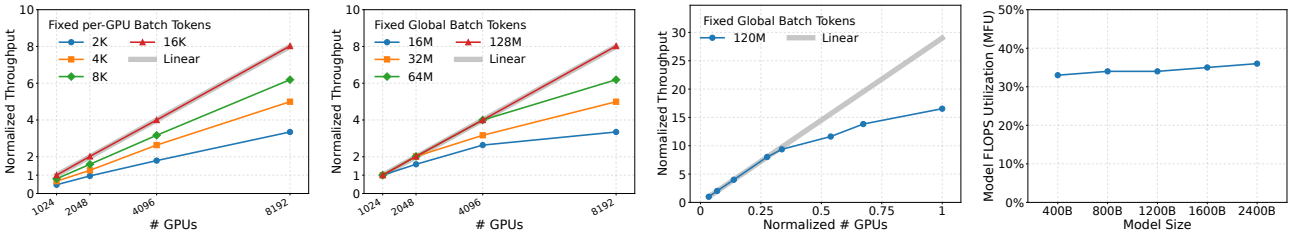
Baselines: We compare veScale-FSDP against state-of-the-art open-source frameworks: DeepSpeed ZeRO v0.17.6 (Rajbhandari et al., 2020), PyTorch 2.7.1 FullyShardedDataParallel (FSDP1) (Zhao et al., 2023), PyTorch 2.7.1 `fully_shard` (FSDP2) (PyTorch, 2024), and Megatron-FSDP. For fairness, all frameworks are configured to use ZeRO-3 with mixed precision (i.e., FP32 master weights and BF16 forward/backward). Unless otherwise specified, veScale-FSDP employs element-wise sharding granularity and is compatible with standard training workflows.

Workloads: For the end-to-end comparison with the baselines (§6.1), we evaluate two state-of-the-art open-source models, Llama-3-70B (Dubey et al., 2024) and GPT-OSS-120B (Agarwal et al., 2025), as well as an internal MoE model. Under weak scaling, each device is statically assigned one batch; the sequence length is 4096 for the dense Llama model and 8192 for the MoE models. We use the AdamW optimizer by default. To avoid out-of-memory (OOM) errors for the baselines on GPT-OSS, we also report results using the SGD optimizer.

6.1 End-to-End Performance

Figure 8 compares the performance of veScale-FSDP against the baselines on the three representative models introduced in the previous section with 1024 GPUs.

Throughput: On the MoE models, veScale-FSDP is 11∼66% faster than all baselines. On Llama-3-70B, veScale-FSDP is 5% faster than DeepSpeed, FSDP1, and FSDP2, and slightly ahead of Megatron-FSDP. The higher throughput arises from optimized communication overlapping, `DBuffer`-based zero-copy collectives, and flexible sharding granularities that avoid padding overhead. In contrast, DeepSpeed emits fragmented collectives (Halilakin, 2024), while FSDP1 exhibits communication bubbles where data movement operations block NCCL progress, underutilizing the network in both systems. FSDP2 relies on the per-parameter DTensor even-sharding format that introduces interleaved Copy-Out after AllGather and interleaved Copy-In before ReduceScatter; together these copies can consume up to 14% of a training iteration and hence reduce throughput. In addition, FSDP1 and FSDP2 do not enforce NCCL buffer address alignment, so collectives may operate on unaligned memory addresses and suffer substantial communication performance degradation in certain cases (Wu et al., 2025). Although Megatron is optimized for zero-copy collectives, its fixed `Shard(0)` sharding granularity, chosen to remain consistent with the upstream DTensor `Shard(0)` semantics for distributed checkpointing, induces 33% buffer padding inflation in MoE models and thus slows collective communication (Megatron, 2025). Our experiments show that veScale-FSDP achieves linear scalability; detailed analysis appears in §6.2.



(a) Weak scaling to 8K GPUs (b) Strong scaling to 8K GPUs (c) Strong scaling (d) Model scaling on 1K GPUs

Figure 9. Scalability of veScale-FSDP. (a) Weak scaling to 8K GPUs. (b) Strong scaling to 8K GPUs. (c) Strong scaling at larger scale with normalized throughput and normalized GPU count. (d) Model scaling on 1K GPUs up to 2.4T parameters.

Memory: Across benchmarks, veScale-FSDP reduces peak *reserved* memory by 16–30%. The memory saving stems from deterministic, batched `DBuffer` memory management: We explicitly manage stream dependencies for predictable memory deallocation, and we batch allocations to reduce fragmentation. By contrast, DeepSpeed and FSDP1 inherit non-deterministic deallocations from PyTorch's implicit `record_stream` mechanism (Gu et al., 2023), which often prevents the caching allocator from reusing buffers, inflating peak reserved memory by 20%. Relative to FSDP2 's per-parameter eager allocation, our batched policy yields a further 12% reduction. Megatron's padding-inflated buffers not only degrade collective efficiency but also raise peak memory by 33% in MoE experiments; its mixed-precision support persists low-precision buffers, consuming 24% more memory than veScale-FSDP in the Llama-3 experiments. Lower reserved memory translates directly into higher end-to-end efficiency: under high memory pressure, the PyTorch caching allocator issues device frees that synchronize with the driver and stall training. In terms of scalability, veScale-FSDP's memory footprint decreases monotonically as the FSDP group size increases and grows only marginally with the replication factor, matching scaling expectations. A notable exception appears with GPT-OSS: FSDP2 trains at 128 devices but OOMs at 256. The per-parameter sharding design in DTensor requires padding to enforce even splits along the sharded dimension; with 128 experts spread over 256 devices, the AllGather buffer effectively doubles, exhausting memory. While FSDP2 allows custom sharding along other dimensions, it requires manual padding and thus doubles the interleaved-copy overhead, making it prohibitively expensive (recall Table 1).

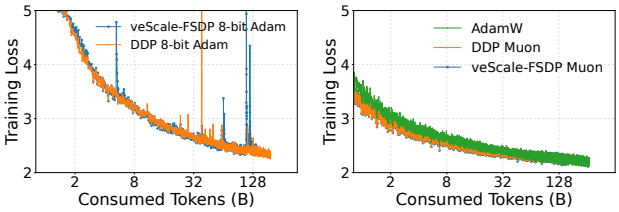
6.2 Scalability and Composability

The flexibility of `RaggedShard` also enables seamless integration with complementary parallelization strategies such as Expert Parallelism (EP) (Lepikhin et al., 2020). Combining these techniques allows veScale-FSDP to efficiently scale training to internal models with up to 2.4T parameters on as many as 10K Hopper GPUs, as shown in Figure 9. Note that we evaluate scalability of MoE, because MoE workloads are often more challenging to scale under FSDP: sparse expert computation lowers per-GPU compute while requiring substantial AllGather/ReduceScatter traffic, making communication and padding overheads more significant.

Weak scaling: Figure 9a presents the weak scaling performance of veScale-FSDP. We train an 800B-parameter MoE internal model on 1K to 8K GPUs while keeping the input size fixed at 2K–16K tokens per GPU. Across all input sizes, veScale-FSDP demonstrates near-linear scalability as the GPU count increases. This is expected since the communication cost of FSDP and the computation cost per GPU remain constant with respect to the number of GPUs, depending only on the model and input sizes. These results confirm the efficiency of veScale-FSDP on large-scale GPU clusters.

Strong scaling: We further evaluate the strong scaling performance of veScale-FSDP by fixing the global batch size to 16M–128M tokens and tuning expert and sequence parallelism configurations for each setting. Figures 9b and 9c show the resulting throughput across different numbers of GPUs. veScale-FSDP scales linearly with a 120M-token global batch up to 10K GPUs, while still delivering a 3.4× throughput gain from 1K to 8K GPUs at a 16M-token global batch. Figure 9c presents the same strong-scaling trend using normalized throughput and normalized GPU count. When the number of GPUs is small, each GPU processes enough tokens to fully overlap communication with computation, yielding near-linear scaling. However, as the GPU count continues to increase, fewer tokens are assigned per GPU per iteration, causing FSDP communication—including parameter AllGather and gradient ReduceScatter—to dominate runtime. To mitigate this overhead, we adopt cross-node Expert Parallelism, which further reduces FSDP communication time. This optimization introduces higher computation cost due to token exchange and reduced kernel efficiency, resulting in the performance drop at very large scales.

Model scaling: We also evaluate model scaling by fixing



(a) 8-bit Adam (b) Distributed Muon

Figure 10. Training convergence with veScale-FSDP on 64 H800 GPUs (FSDP size 64) for 8-bit Adam and distributed Muon.

the GPU count to 1K and increasing the model size from 400B to 2.4T parameters. With model sparsity constant and 8K training tokens per GPU, we scale both depth (number of layers) and width (intermediate dimensions) proportionally. Figure 9d reports the effective Model FLOPS Utilization (MFU) per GPU as model size grows. Enabled by efficient memory management of `DBuffer` (§5), veScale-FSDP can train 2.4T-parameter models on only 1K GPUs without any performance degradation. In fact, MFU slightly improves with larger models due to the increased compute intensity and better utilization of GPU resources.

6.3 8-bit Adam and Muon Optimizer

We demonstrate the flexibility of `RaggedShard` DTensor using two examples: the 8-bit Adam optimizer and the distributed Muon optimizer.

8-bit Adam optimizer. 8-bit Adam (Dettmers et al., 2022) applies *block-wise* INT8 quantization to the gradient statistics, substantially reducing optimizer-state memory. To enable 8-bit Adam, veScale-FSDP exposes an `orig_param_policy` interface that lets users set the quantization *granularity* per parameter. In our setup, we use 32×32 blocks and assign matrix parameters to 32-row block granularity. With this layout, each device quantizes its local shard independently without any communication, and block boundaries are perfectly preserved by `RaggedShard`. In contrast, existing FSDP systems do not natively track such block boundaries, so enabling block-wise 8-bit Adam often requires intrusive system changes or manual collectives to exchange quantization metadata, incurring both complexity and overhead.

We implement 8-bit Adam using veScale-FSDP with few lines of code and provide the evaluation in Figure 10a. We compare 8-bit Adam under distributed data parallelism (DDP) and with veScale-FSDP. The loss curves track closely, with occasional spikes characteristic of reduced-precision optimizer states. The small difference stems from the gradient-reduction schedule: DDP uses bucketed AllReduce, whereas veScale-FSDP performs layer-wise ReduceScatter. (Note that the loss curves in Figure 10 are not directly comparable: for 8-bit Adam, we use a smaller learning rate to mitigate overflow/underflow in reduced precision.)

Algorithm 2 `RaggedShard` Distributed Muon

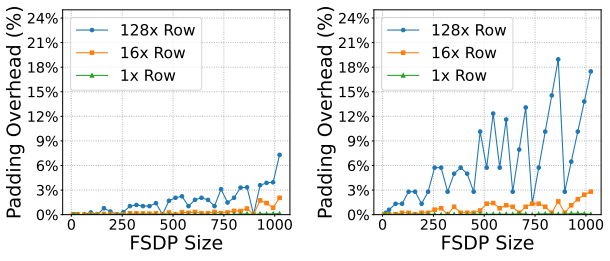
1: **for all** $\mathbf{w}$ in 2D parameter tensors **do**
2: $\quad \mathbf{g} \leftarrow \mathrm{grad}(\mathbf{w})$
3: $\quad \mathbf{u} \leftarrow \textsc{MomentumUpdate}(\mathbf{g}, \mathbf{m})$
4: $\quad p \leftarrow \mathrm{placement}(\mathbf{u})$ // *original DTensor placement*
5: $\quad$ // *Choose compute device via load balancing*
6: $\quad r \leftarrow \textsc{SelectRoot}()$
7: $\quad$ // *Unshard to root via redistribute*
8: $\quad \mathbf{o} \leftarrow \textsc{Redistribute}(\mathbf{u}, \texttt{RaggedShard}(r))$
9: $\quad$ // *Muon update: Newton–Schulz on full tensor.*
10: $\quad \mathbf{o} \leftarrow \textsc{NewtonSchulz}(\mathbf{o})$
11: $\quad$ // *Redistribute update back.*
12: $\quad \mathbf{o} \leftarrow \textsc{Redistribute}(\mathbf{o}, p)$
13: $\quad \mathbf{w} \leftarrow \mathbf{w} - \eta\, \mathbf{o}$
14: **end for**

Distributed Muon optimizer. The matrix-sign preconditioner (e.g., Newton–Schulz) of Muon requires the *full* 2D parameter matrix with its original shape. Algorithm 2 sketches the distributed Muon optimizer enabled by `RaggedShard`. Thanks to `RaggedShard`'s capability to support uneven sharding, users can write Muon's parameter-gather step in a clean SPMD way: after redistribution, only the root rank holds the full 2D parameter, so the Newton–Schulz update becomes a no-op on other ranks. As lines 5–8 show, the algorithm selects a root via load balancing and unshards to it using the standard DTensor `redistribute` with `RaggedShard` placement. Lines 9–10 run the Muon matrix iteration only on the root that holds the full tensor. Finally, lines 11–13 redistribute the update back to the original device and apply it. Therefore, users do not need to handle the complex logic of communication and can further overlap communication with computation via asynchronous `redistribute`. In addition, our optimized Muon reaches 47.3% MFU on 256 Hopper GPUs by exploiting the communication-computation overlapping and using `torch.compile` to further increase compute density.

We implement distributed Muon using veScale-FSDP with few lines of code and provide the evaluation in Figure 10b. We compare the loss curves of Muon with AdamW: the two Muon runs (veScale-FSDP and DDP) match closely, and Muon converges faster than AdamW, stabilizing around 0.01 lower loss after training $\sim$80B tokens, which is consistent with prior results (Wen et al., 2025).

6.4 Planning Quality

A major design objective of the planning algorithm (Algorithm 1) is to enable arbitrary granularity of `RaggedShard`, while minimizing padding overhead and thus reducing com-



(a) DeepSeek-V3-671B (b) GPT-OSS-120B

Figure 11. Padding overhead (extra padding bytes over total parameter size) of `RaggedShard` communication. Lines compare the sharding granularities (1×/16×/128× parameter row size) versus FSDP sharding size (number of GPUs).

Table 2. Component ablation for 8-bit Adam, reported as normalized throughput after disabling each component independently. N/A indicates that the corresponding configuration is not meaningfully runnable without intrusive modifications to the model or optimizer code or manual management of custom collectives.

veScale-FSDP Component	Normalized Throughput
Combined	100.0%
Disable `DBuffer` only	92.8%
Disable Planning Algorithm only	65.4%
Disable `RaggedShard` only	N/A

munication volume. The quality of the planning algorithm can be directly evaluated by padding size. We evaluate it by benchmarking DeepSeek-V3-671B and GPT-OSS-120B across varying device counts. Following a DeepSeek-style quantization scheme, we quantize only the FFN weights (most parameters) and sweep the row granularity of expert-MLP matrices over 128, 16, 1. The 128-row setting reproduces DeepSeek's 128×128 tiling (i.e., weights can be sliced into 128×128 blocks). We then report the resulting relative padding ratios and analyze the root cause of the extra padding.

Figure 11a and 11b show that with 1× and 16× row granularities, veScale-FSDP keeps padding overhead less than 3% across all FSDP sizes for both models. With 128× rows, DeepSeek-V3 remains mostly below 3% with mild growth, whereas GPT-OSS exhibits step-like fluctuations with spikes up to 18%. GPT-OSS fuses all experts into a single parameter tensor, whereas DeepSeek-V3 materializes each expert as a separate parameter; this enables per-expert padding between MLPs and thus relaxes the global padding constraint. The fluctuation behavior is expected: each matrix must be partitioned across the shard group in discrete quanta determined by (i) the granularity unit (e.g., rows) and (ii) NCCL's even-input alignment for high-performance collectives. Effective shard sizes are therefore rounded up to the least common multiple of these granularities; when the group size crosses a multiple, the per-device shard size jumps, producing the observed spikes.

We also evaluate the overhead of the planning algorithm itself: the algorithm runtime is less than 0.3 seconds across all experiments, which is one-time and negligible in distributed training initialization.

Based on Figure 11 and our experience, a practical guideline is to avoid excessively large FSDP group sizes and use hierarchical parallelism (e.g., HSDP) to scale to larger GPU counts. In practice, we select the FSDP group size by offline simulation to minimize LCM-induced rounding for a given model and `RaggedShard` granularity. When model dimensions remain configurable, choosing hidden sizes divisible by small composite factors rather than large co-prime dimensions can further reduce padding overhead.

6.5 Performance Breakdown

To quantify the benefit of each component, we ablate veScale-FSDP by disabling one component at a time and report the resulting throughput, normalized to the full system. We run this study on 32 GPUs when training a GPT-OSS-style model with 8-bit Adam.

Table 2 shows that `DBuffer` and the planning algorithm account for most of the realized speedups: disabling them reduces throughput to 92.8% and 65.4%, respectively. In contrast, `RaggedShard` is not just an optimization; it is the abstraction that makes block-wise 8-bit Adam usable without intrusive model/optimizer changes or hand-written collectives. Specifically:

- **`DBuffer`.** Disabling `DBuffer` drops throughput by 7.2%, reflecting the Copy-In/Copy-Out overhead around collectives when communication buffers require copying.
- **Planning Algorithm.** Disabling the planning drops throughput by 34.6% because quantization blocks are no longer guaranteed to be fully contained within a device's local shard. The system then falls back to DTensor redistribution to assemble the required optimizer states before per-block quantization, incurring substantial extra communication overhead.
- **`RaggedShard` DTensor.** Disabling `RaggedShard` makes it effectively non-runnable: users must either (i) carefully change every model and optimizer tensor so that 32×32 block boundaries align with shard boundaries, or (ii) manually implement complex collectives (e.g., per-block metadata exchange and state gathering) to recover block-wise semantics. We therefore report this setting as N/A to indicate it is not meaningfully usable.

7 LESSONS LEARNED

During the deployment of veScale-FSDP for real industrial workloads that use more than 10K GPUs, we summarize the key lessons we have learned.

Lesson-1: Small-scale workloads can predict large-scale performance. The performance of FSDP-based workloads can be accurately estimated using each layer's computation time and FSDP communication time. Computation occurs entirely within each GPU, and FSDP communication time remains largely unchanged when the number of GPUs increases. This observation is validated by our weak scaling experiments (§6). In practice, we profile the performance of veScale-FSDP on around 64 GPUs and extrapolate to thousands of GPUs, achieving similar results.

This extrapolation assumes that the profiling run exercises network behavior similar to the target scale: comparable network topology, identical collective algorithms/protocols, and a sufficiently large workload to reach bandwidth saturation. To further improve predictability at large scales, we use additional parallelization (e.g., HSDP/EP) to cap the collective group size, preventing excessively large collectives whose latency can vary more.

Lesson-2: Design system abstractions on the shoulders of giants. DTensor provides a powerful abstraction that already supports a wide range of parallelization techniques. By designing new abstractions on top of DTensor, we can seamlessly integrate existing parallelization strategies. In our work, `RaggedShard` is implemented as an optional placement on DTensor, enabling easy collaboration of established infrastructure such as Tensor and Expert Parallelism, as well as mature training tools like distributed checkpointing (PyTorch Team, 2025). This approach minimizes engineering effort while contributing to a shared ecosystem that benefits the broader community. In fact, `RaggedShard` has already appeared as a planned feature on the official roadmap (PyTorch Team, 2026) of PyTorch.

Lesson-3: Decoupled model definition with system optimization matters. The rapid evolution of model architectures demands frequent updates to model definitions. However, existing frameworks such as Megatron-LM tightly couple system-level parallelization optimizations with model code, making it difficult for researchers to modify or extend architectures. In veScale-FSDP, we decouple model definition from the system framework, allowing researchers to focus on model design while maintaining linear scalability across up to 10K GPUs. This separation greatly simplifies model development and accelerates architectural innovation.

8 CONCLUSION

veScale-FSDP is a scalable training system that combines high flexibility with high performance through the `RaggedShard` abstraction and a structure-aware planning algorithm that maximizes GPU utilization. Experiments demonstrate that veScale-FSDP seamlessly integrates with emerging techniques such as Muon optimizers and significantly outperforms existing systems, achieving 5∼66% higher throughput and 16∼30% lower memory usage, while scaling efficiently to tens of thousands of GPUs.

9 ACKNOWLEDGMENTS

veScale-FSDP would not have been possible without the tremendous support and collaboration of our teammates and colleagues. We sincerely thank them (in no particular order; this list is not exhaustive):

- veScale members: Hongrui Zhan, Ziyi Zhang, Hao Feng
- ByteDance teammates: Jianyu Jiang, Chenyuan Wang, Cesar Andres Stuardo Moraga, Juntao Zhao, Bin Jia, Chengye Li, Zhongkai Zhao, Shixiong Zhao, Tiantian Fan, Hanshi Sun, Wenlei Bao, Shixun Wu, Zhekun Zhang, Yanbo Liang, Li-wen Chang, Jun Wang, Cheng Li, Li Han, Heng Zhang, Zhenbo Sun, Bo Liu, Xiaonan Nie, Ru Zhang, Hao Gong, Zuquan Song, Yucheng Nie, Jiawei Wu, Hongpeng Guo, Xinyi Di

Equally important, we thank everyone on the TorchTitan team and Edward Z. Yang for the insightful discussions and collaboration within the open-source community.

REFERENCES

Agarwal, S., Ahmad, L., Ai, J., Altman, S., Applebaum, A., Arbus, E., Arora, R. K., Bai, Y., Baker, B., Bao, H., et al. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*, 2025.

Dettmers, T., Lewis, M., Shleifer, S., and Zettlemoyer, L. 8-bit optimizers via block-wise quantization. In *International Conference on Learning Representations*, 2022.

Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., et al. The llama 3 herd of models. *arXiv e-prints*, pp. arXiv–2407, 2024.

Garey, M. R. and Johnson, D. S. Complexity results for multiprocessor scheduling under resource constraints. *SIAM journal on Computing*, 4(4):397–411, 1975.

Gu, A., Feng, W., and Zhao, Y. [rfc] per-parameter-sharding fsdp, 2023. URL https://github.com/pytorch/pytorch/issues/114299.

Gupta, V., Koren, T., and Singer, Y. Shampoo: Preconditioned stochastic tensor optimization. In *International Conference on Machine Learning*, pp. 1842–1850. PMLR, 2018.

Halilakin. Deepspeed is slower than fsdp, 2024. URL https://github.com/deepspeedai/DeepSpeed/issues/5047#issuecomment-1926275502.

Jiang, Z., Lin, H., Zhong, Y., Huang, Q., Chen, Y., Zhang, Z., Peng, Y., Li, X., Xie, C., Nong, S., et al. {MegaScale}: Scaling large language model training to more than 10,000 {GPUs}. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pp. 745–760, 2024.

Jordan, K., Jin, Y., Boza, V., You, J., Cesista, F., Newhouse, L., and Bernstein, J. Muon: An optimizer for hidden layers in neural networks, 2024. URL https://kellerjordan.github.io/posts/muon/.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Lepikhin, D., Lee, H., Xu, Y., Chen, D., Firat, O., Huang, Y., Krikun, M., Shazeer, N., and Chen, Z. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

Li, Y., Wan, C., Lin, Z., Zhu, H., Yang, J., Song, Z., Di, X., Wu, J., Shu, H., Bao, W., Peng, Y., Lin, H., and Chang, L.-W. veScale: Consistent and Efficient Tensor Programming with Eager-Mode SPMD, 2025. URL https://arxiv.org/abs/2509.07003.

Liu, A., Feng, B., Xue, B., Wang, B., Wu, B., Lu, C., Zhao, C., Deng, C., Zhang, C., Ruan, C., et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

Ma, Q., Zheng, Y., Shi, Z., Zhao, Z., Jia, B., Huang, Z., Lin, Z., Li, Y., Yang, J., Peng, Y., Zhang, Z., and Liu, X. VeOmni: Scaling Any Modality Model Training with Model-Centric Distributed Recipe Zoo, 2025. URL https://arxiv.org/abs/2508.02317.

Megatron. Mcore custom fully sharded data parallel (fsdp). Technical report, 2025.

NCCL, N. Regarding the allgather bandwidth with different byte alignment under different protocols, 2025a. URL https://github.com/NVIDIA/nccl/issues/413.

NCCL, N. Nccl: Collective operations, 2025b. URL https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/collectives.html.

PyTorch. Fully sharded data parallel (fsdp2). Technical report, 2024.

PyTorch. Pytorch jaggedtensor, 2025a. URL https://docs.pytorch.org/FBGEMM/fbgemm_gpu/overview/jagged-tensor-ops/JaggedTensorOps.html.

PyTorch. Pytorch nestedtensor, 2025b. URL https://docs.pytorch.org/docs/main/nested.html.

PyTorch Team. Distributed checkpoint, 2025. URL https://docs.pytorch.org/docs/stable/distributed.checkpoint.html#distributed-checkpoint-torch#-distributed-checkpoint.

PyTorch Team. Meta pytorch team 2026 h1 roadmaps, 2026. URL https://dev-discuss.pytorch.org/t/meta-pytorch-team-2026-h1-roadmaps.

Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16. IEEE, 2020.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Smith, S., Patwary, M., Norick, B., LeGresley, P., Rajbhandari, S., Casper, J., Liu, Z., Prabhumoye, S., Zerveas, G., Korthikanti, V., et al. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv preprint arXiv:2201.11990*, 2022.

Team, G., Georgiev, P., Lei, V. I., Burnell, R., Bai, L., Gulati, A., Tanzer, G., Vincent, D., Pan, Z., Wang, S., et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*, 2024.

Team, K., Bai, Y., Bao, Y., Chen, G., Chen, J., Chen, N., Chen, R., Chen, Y., Chen, Y., Chen, Y., et al. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025.

TensorFlow. Tensorflow ragged tensors, 2025. URL https://www.tensorflow.org/guide/ragged_tensor.

The PyTorch Team. PyTorch DTensor (Distributed Tensor). https://pytorch.org/docs/stable/distributed.tensor.html, 2024.

Wen, K., Hall, D., Ma, T., and Liang, P. Fantastic pretraining optimizers and where to find them. *arXiv preprint arXiv:2509.02046*, 2025.

Wu, B., Cui, W., Curino, C., Interlandi, M., and Sen, R. Terabyte-scale analytics in the blink of an eye. *arXiv preprint arXiv:2506.09226*, 2025.

Xu, J. FSDP & CUDACachingAllocator. https://dev-discuss.pytorch.org/t/fsdp-cudacachingallocator-an-outsider-newb-perspective/1486, 2024. PyTorch Dev Discuss.

Xu, Y., Lee, H., Chen, D., Hechtman, B., Huang, Y., Joshi, R., Krikun, M., Lepikhin, D., Ly, A., Maggioni, M., et al. Gspmd: general and scalable parallelization for ml computation graphs. *arXiv preprint arXiv:2105.04663*, 2021.

Zhao, Y., Gu, A., Varma, R., Luo, L., Huang, C.-C., Xu, M., Wright, L., Shojanazeri, H., Ott, M., Shleifer, S., et al. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*, 2023.

Zhao, Y., Gu, A., Varma, R., Luo, L., Huang, C.-C., Xu, M., Wright, L., Shojanazeri, H., Ott, M., Shleifer, S., et al. Fsdp1 post backward reduce, 2025. URL https://github.com/pytorch/pytorch/blob/a4925c0ce004cf883fdd1b248d71676769524934/torch/distributed/fsdp/_runtime_utils.py#L695C1-L773C1.