PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel

Yanli Zhao
Meta AI
yanlizhao@meta.com

Andrew Gu
Meta AI
andgu@meta.com

Rohan Varma
Meta AI
rvarm1@meta.com

Liang Luo
Meta AI
liangluo@meta.com

Chien-Chin Huang
Meta AI
chienchin@meta.com

Min Xu
Meta AI
m1n@meta.com

Less Wright
Meta AI
less@meta.com

Hamid Shojanazeri
Meta AI
hamidnazeri@meta.com

Myle Ott
Meta AI
myleott@gmail.com

Sam Shleifer
Meta AI
sshleifer@gmail.com

Alban Desmaison
Meta AI
albandes@meta.com

Can Balioglu
Meta AI
balioglu@meta.com

Pritam Damania
Meta AI
pritam.damania@gmail.com

Bernard Nguyen
Meta AI
bernardn@meta.com

Geeta Chauhan
Meta AI
gchauhan@meta.com

Yuchen Hao
Meta AI
haoyc@meta.com

Ajit Mathews
Meta AI
amath@meta.com

Shen Li
Meta AI
shenli@meta.com

ABSTRACT

It is widely acknowledged that large models have the potential to deliver superior performance across a broad range of domains. Despite the remarkable progress made in the field of machine learning systems research, which has enabled the development and exploration of large models, such abilities remain confined to a small group of advanced users and industry leaders, resulting in an implicit technical barrier for the wider community to access and leverage these technologies. In this paper, we introduce PyTorch Fully Sharded Data Parallel (FSDP) as an industry-grade solution for large model training. FSDP has been closely co-designed with several key PyTorch core components including Tensor implementation, dispatcher system, and CUDA memory caching allocator, to provide non-intrusive user experiences and high training efficiency. Additionally, FSDP natively incorporates a range of techniques and settings to optimize resource utilization across a variety of hardware configurations. The experimental results demonstrate that FSDP is capable of achieving comparable performance to Distributed Data Parallel while providing support for significantly larger models with near-linear scalability in terms of TFLOPS.

PVLDB Reference Format:
Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. PVLDB, 16(12): 3848-3860, 2023.
doi:10.14778/3611540.3611569

Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/pytorch/pytorch/blob/main/torch/distributed/fsdp/fully_sharded_data_parallel.py/.

1 INTRODUCTION

The magnitude of neural network models is growing at an unprecedented rate, facilitating breakthroughs across a wide spectrum of domains. Upon inception, the 175-billion-parameter GPT-3 [3] model set a new record for almost all Natural Language Processing tasks. The product applications constructed on top of GPT models [23] have quickly demonstrated their potential to revolutionize the entire industry. Modern large scale recommendation models [19, 33] can reach beyond 1 trillion parameters, replete with rapidly growing dense layer components. These models power applications that serve multi-billions of users every day. As large neural networks continue to push the limits of science and technology, an industry-grade tool to simplify the training of such models with high efficiency would help expedite the progress.

In recent years, the community has introduced and investigated numerous advanced methodologies to enlarge neural network models. Pipeline parallelism [6, 8, 11, 15, 20] partitions a model instance into stages and distributes stages across multiple devices, where activations and gradients are communicated across stage boundaries. Tensor parallelism [9, 21, 31, 32] shards model parameters, conducts partial computation on individual devices and communicates activations at required layer boundaries. Zero-Redundancy parallelism [27, 28, 30] shards parameters as well but communicates parameters on-demand to recover their unsharded form and executes the model as if it were replicated on every device. The aforementioned techniques have served as the fundamental building

doi:10.14778/3611540.3611569

blocks to enable the training of large neural networks across various applications. Nevertheless, two challenges still persist. Firstly, some of these methods are tightly integrated with specific model architectures, which hinder them from being utilized as a generic solution for training large models. Secondly, some of these techniques are built on top of rapidly-evolving internal interfaces of underlying machine learning frameworks, which become vulnerable to changes in framework implementations. Therefore, it is more robust and efficient to have a native solution co-designed with the core functionalities of machine learning frameworks. Additionally, constructing such a solution in a composable and customizable manner could potentially facilitate the community's future innovations as well.

This paper presents PyTorch [24] Fully Sharded Data Parallel (FSDP), which enables the training of large-scale models by sharding model parameters. The FSDP algorithm is motivated by the ZeroRedundancyOptimizer [27, 28] technique from DeepSpeed but with a revised design and implementation that is aligned with the other components of PyTorch. FSDP breaks down a model instance into smaller units and then flattens and shards all of the parameters within each unit. The sharded parameters are communicated and recovered on-demand before computations, and then they are immediately discarded afterwards. This approach ensures that FSDP only needs to materialize parameters from one unit at a time, which significantly reduces peak memory consumption. The design and implementation of FSDP faces the following challenges.

- **User Experience** is critical for achieving broad adoption. When working on prior PyTorch distributed training features such as `DistributeDataParallel` (DDP) [14], we observed that aligning the user experience of distributed training with that of local training can significantly lower the learning barrier. Techniques like DDP require the model to be replicated on every device, which implies that the entire model can be constructed on the target device. However, although FSDP can easily adopt DDP's API design, large models might not fit into one GPU device and therefore cannot even be initialized efficiently.
- **Hardware Heterogeneity** often exists in modern GPU clusters, whereby interconnects are partitioned into high-bandwidth islands within each machine and low-bandwidth mesh across machines. Additionally, there may be further hierarchical structures at the rack or pod levels. Consequently, the design of FSDP must accommodate such heterogeneity and optimize accordingly.
- **Resource Utilization** is usually tightly linked with capital and operational expenditures, especially for companies that depend on large GPU clusters to power their mission-critical systems. To ensure that GPU devices remain fully utilized during distributed training, it is essential to minimize downtime caused by non-computational operations.
- **Memory Planning** plays a crucial role in large model training. PyTorch makes GPU memory block allocation efficient and transparent through caching. Frequent memory defragmentations can significantly slow down training, which becomes particularly acute when working with large models. In such scenarios, practitioners typically seek to saturate GPU memory as much as possible to accommodate the largest batches or models. However, operating near GPU memory capacity significantly increases the chance to trigger defragmentations.

FSDP tackles the aforementioned challenges through a variety of techniques. Firstly, to improve user experience, FSDP introduces deferred initialization that allows users to create a model instance on a dummy device and record operations invoked during initialization. Then, the model can be initialized and sharded unit by unit by replaying the recorded operations on a real GPU device. With this technique, FSDP can provide similar user experiences as local training, while effectively scaling large models. Secondly, FSDP offers configurable sharding strategies that can be customized to match the physical interconnect topology of the cluster to handle hardware heterogeneity. Thirdly, although parameter sharding design inevitably inserts communications, which might block computations and introduces bubbles during execution, FSDP can squeeze out bubbles using an abundant set of tools to aggressively overlap communication with computation through operation reordering and parameter prefetching. Lastly, FSDP optimizes memory usage by prudently restricting the amount of blocks allocated for inflight unsharded parameters and suspending CPU execution if necessary.

We evaluated the performance of FSDP on various models including popular language models and recommendation system models, utilizing up to 512 80GB A100 GPUs. The experiments showed that FSDP can achieve similar performance to that of DDP on small models. Beyond that FDSP can facilitate significantly larger models with near-linear scalability in terms of TFLOPS. FSDP is currently a beta feature as of PyTorch 2.0 release, and has been battle-tested by both industrial and research applications.

To simplify presentation, the rest of this paper uses FSDP to refer to the techniques in general and `FullyShardedDataParallel` to denote the Python implementation. The remainder of the paper is organized as follows. Section 2 introduces background on some popular distributed training techniques. Section 3 and Section 4 elaborate system design and implementation details. Evaluations are presented in Section 5. Section 6 surveys related work, and Section 7 discusses topics related to FSDP but falls outside of FSDP core. Finally, Section 8 concludes the paper.

2 BACKGROUND

PyTorch [24] has emerged as a fundamental cornerstone for a plethora of machine learning endeavors. PyTorch stores values in `Tensor` objects, which are versatile n-dimensional arrays featuring a rich set of data manipulation operations. Every `Tensor` object has an associated storage that is allocated on a specific device. When `Tensor`s only represent simple transformations such as `reshape` and `split`, they can share the same underlying storage. Each `Module` describes a transformation from input to output values, and its behavior during the forward pass is specified by its `forward` member function. Such a module may feature `Tensor` objects as parameters, with the `Linear` module being an example that contains both `weight` and `bias` parameters. During the forward pass, the `Linear` module applies these parameters to the input to produce the output by means of multiplication and addition operations, respectively.

As both the data size and model complexity continue to escalate at a staggering pace, the need for an industry-grade distributed training framework becomes increasingly imperative for applications built on top of PyTorch. This section elucidates the trajectory of PyTorch's distributed training capabilities.

2.1 Model Replication

Model replication approaches are designed to tackle high-volume datasets by scaling out and distributing computations across multiple devices. `DistributedDataParallel` (DDP) [14] is the first end-to-end distributed training feature in PyTorch that falls into this category. DDP's adoption has been extensive, spanning both the academic and industrial domains.

DDP maintains a model replica on each device and synchronizes gradients through collective `AllReduce` operations in the backward pass, thereby ensuring model consistency across replicas during training. To expedite training, DDP overlaps gradient communication with backward computation, facilitating concurrent workload executions on diverse resources. However, one conspicuous limitation is that DDP requires all model parameters, gradients, and optimizer states to fit in the memory of one GPU device. Consequently, DDP is inadequate for supporting large models, which are critical for cutting-edge machine learning breakthroughs. For example, when training models with more than one billion parameters using a 40GB GPU device, DDP will likely encounter out-of-memory errors on each device.

2.2 Model Partitioning

As the size of models grow, they may no longer fit in a single GPU device. In such cases, a viable solution is to partition the model into smaller components and distribute them across multiple devices. Both pipeline parallelism [8] and Tensor RPC [25] are along this direction. Pipeline parallelism involves breaking a sequence of layers into stages and feeding inputs to different stages in a pipelined fashion to optimize resource utilization. On the other hand, Tensor RPC provides a lower-level toolkit that enables arbitrary computations to be executed on remote devices. While both techniques are capable of scaling large models across multiple devices, they either limit the model to a sequence of stages or require modifications to the model authoring code to insert remote computations, which can pose a significant obstacle to users' adoption. Moreover, many industrial training infrastructures only support the single-program multi-data paradigm, which necessitates a simpler entry point to handle large models.

2.3 Model Sharding

In addition to partitioning, sharding the parameters of a model can also help reduce its memory footprint and support models with sizes beyond the memory capacity of a single GPU device. After sharding models, each rank only holds a shard of the model parameters, which prevents it from performing the same computations as local training. To guarantee correctness, the training process needs to employ one or both of the following techniques:

- Perform computations with parameter shards and **communicate activations** accordingly. With this approach, ranks never need to fully materialize any parameter. However, each communication will appear in the critical path as it is inserted between two consecutive and dependent computation operations. As a result, this communication cannot easily overlap with computations, unless non-dependent computations or computations from other iterations can be re-ordered to overlap with communication.
- Perform the same computation as local training by **communicating parameter** on-demand before computations. Since parameter communications do not have any data dependency on preceding computations, they can overlap with the preceding computations performed in the same forward or backward pass. However, this approach requires that the on-demand communicated parameters could be fully materialized and could fit in the memory of a single GPU device.

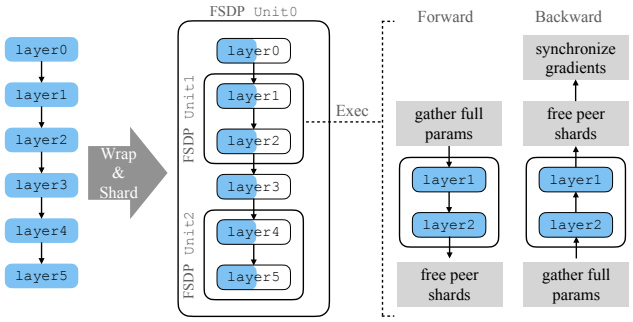


Figure 1: FSDP Algorithm Overview

FSDP falls into the second category of communicating parameters. Based on our observations and experiments, this approach is sufficient to support the vast majority of large model applications today and in the near future. It is worth noting that if the requirement of fully materializing each parameter unit on GPU becomes a blocker, we can further combine both techniques to support such use cases.

3 SYSTEM DESIGN

Fully Sharded Data Parallel (FSDP) is capable of scaling to accommodate large models that may not fit in a single GPU device by sharding the dense parameters. More specifically, FSDP decomposes the model instance into smaller units and handles each unit independently. During forward and backward computation, FSDP only materializes unsharded parameters and gradients of one unit at a time, and otherwise, it keeps parameters and gradients sharded. Throughout the training loop, the optimizer states are kept sharded. The memory requirements for FSDP are proportional to the size of the sharded model plus the size of the largest fully-materialized FSDP unit.

Figure 1 demonstrates the overall workflow using a simple six layer model. Suppose FSDP decomposes the model into three parts, namely, `[layer0, layer3]`, `[layer1, layer2]`, and `[layer4, layer5]`. The decomposition behavior can be controlled by user-defined functions. FSDP then wraps each of these three parts into one FSDP unit and shards parameters accordingly. To ensure correctness, FSDP needs to recover the unsharded parameters before corresponding

computations. Let us consider FSDP `unit1` that contains `[layer1, layer2]` to explain this process. Before forward computation enters `layer1`, FSDP collects the unsharded parameters for `layer1` and `layer2` by gathering shards from other peer ranks. With the unsharded parameters, FSDP runs the local computation of those layers and then frees the peer shards it just collected to reduce memory footprint. Therefore, during the entire forward pass, FSDP only needs to fully materialize one unit at a time, while all other units can stay sharded. Similarly, during the backward computation, FSDP `unit1` recovers the unsharded parameters for `layer1` and `layer2` before backward reaches `layer2`. When the autograd engine finishes the backward computation of these two layers, FSDP frees the peer shards and launches `ReduceScatter` to reduce and shard gradients. Hence, after backward computation, each rank only keeps a shard of both parameters and gradients.

FSDP offers a wide spectrum of optimizations and knobs to account for diverse model structures and hardware capabilities. The remainder of this section delves further into the intricacies of model initialization, sharding strategies, communication optimizations, and memory management, which are all critical components of FSDP's underlying design.

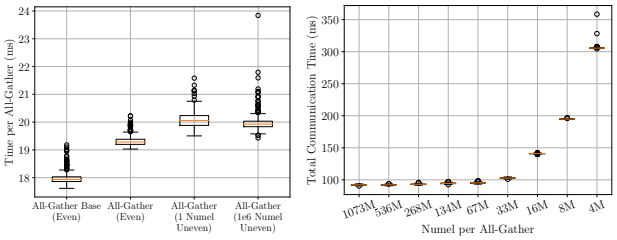
3.1 Model Initialization

Before the advent of FSDP, PyTorch mandated the full materialization of the entire model instance on one device. Although users can allocate different sub-modules to different devices, this would require modifying the model source code, which may not be feasible, particularly if model authors and application developers belong to different parties. To facilitate a smooth transition from local to distributed training, FSDP must effectively aid in the materialization and initialization of a massive model, which poses two challenges:

- How to create a model instance without materializing any tensor storage, postponing initialization until a storage on a concrete device is attached to the tensor.
- How to ensure accurate initialization of model parameters in line with the user's implementation, even when the model is too large to fit on a single GPU.

To overcome the first challenge, we have introduced a mechanism called *deferred initialization*, which involves the allocation of model parameter tensors on a simulated or "fake" device. During this process, all initialization operations performed on the tensor are recorded. Subsequently, when the tensor is moved from the "fake" device to a GPU device, all recorded operations are automatically replayed. By adopting this technique, users can generate a model instance from any third-party library without allocating any GPU memory blocks, while still accurately capturing their parameter initialization implementations.

As illustrated in Figure 1, once the FSDP has wrapped the model, it is evenly distributed across all GPUs, with each device holding only one shard in its memory. Therefore, in order to address the second challenge, each rank should ideally only materialize and initialize the shard that it owns. However, this is not always practical, since we cannot predict what initialization logic the user will implement in the model `init` method. The initialization logic may rely on having a unsharded parameter on the device, which makes it impossible to shard the initialization. Consequently, FSDP must prepare the unsharded parameters before executing `Tensor` initialization operations and simultaneously reduce the memory footprint. Given that sharding initialization is unsafe, FSDP applies the same approach as how it handles model forward and backward passes, *i.e.*, initialize one FSDP unit at a time and shard the unit before moving on to the next one. When combined with deferred initialization, FSDP traverses the fake device model instance to decompose it into FSDP units, moves one unit to a GPU device at a time, and replays the recorded initialization operations for tensors in that FSDP unit.



(a) Uneven Input Sizes (b) Reducing Input Size

Figure 2: Communication Efficiency vs. Input Size

3.2 Sharding Strategies

The *sharding strategy* is an important element in FSDP that plays a significant role in determining the memory footprint and communication overhead. FSDP offers a variety of sharding strategies, ranging from fully replicated to fully sharded. To generalize these sharding strategies, we introduce the *sharding factor F* as the number of ranks over which parameters are sharded. By setting the sharding factor to 1, FSDP fully replicates the model and simplifies to vanilla data parallelism that uses `AllReduce` for gradient reduction. By setting the sharding factor equal to the number of devices (*i.e.*, global world size W), FSDP fully shards the model, with each device only holding $\frac{1}{W}$ of the model. Hybrid sharding occurs when the sharding factor ranges between 1 and W. The remainder of this section focuses on full sharding and hybrid sharding since the full replication strategy is similar to the existing DDP [14].

3.2.1 *Full Sharding.*

The *full sharding* strategy leads to the lowest memory footprint but incurs the most communication overhead, for example, *full sharding* has 1.5x communication overhead and volume over DDP if using bandwidth optimal ring algorithm. Therefore, FSDP must carefully organize communications to maximize its efficiency under this strategy.

We conducted two sets of experiments to understand the impact of input size on collective communication efficiency. Results are shown in Figure 2, which helped identify two ingredients for efficiencies:

(1) **Even Input Size:** The Nvidia NCCL [22] library offers efficient collective implementations for all-gather and reduce-scatter that require *even* input tensor sizes across ranks.

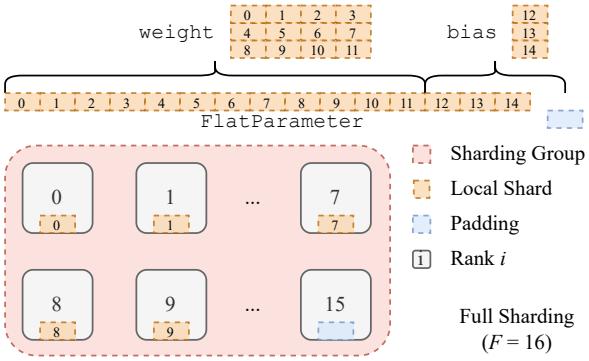


Figure 3: Full Sharding Across 16 GPUs

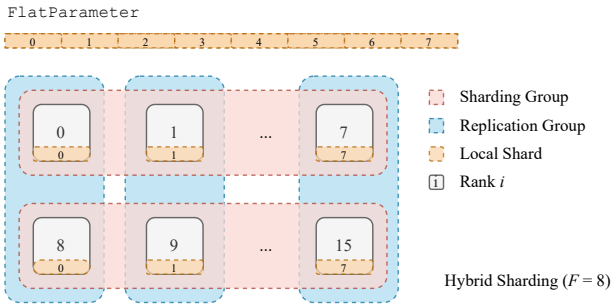


Figure 4: Hybrid Sharding on 16 GPUs: GPUs are configured into 2 sharding groups and 8 replication groups

(2) **Larger Input Size:** For fixed communication volume, batching data and issuing fewer collectives improves performance by avoiding the collectives' launch overhead and increasing network bandwidth utilization.

For (1), NCCL's `AllGather` API requires even input tensor size and writes outputs into one single tensor. PyTorch's `ProcessGroup` wraps the NCCL API and enhances it by supporting uneven input tensor sizes across ranks and allowing users to provide a list of output tensors. The flexibility comes with an efficiency trade-off, as shown in Figure 2 (a). We use `All-Gather Base` to denote NCCL's `AllGather` behavior, and `All-Gather` to denote the one that takes a list of tensors as outputs. The latter incurs additional copies between the individual output tensors and the consolidated single large output tensor before and after the communication. Moreover, for uneven inputs, `ProcessGroup` mimics `AllGather`'s behavior using group `Broadcast`, which is slower than `All-Gather Base`. In the experiments, we created artificial unevenness by moving 1 element and $1e6$ elements from rank 1 to rank 0 respectively. The results show that the `All-Gather Base` with even input size achieved highest efficiency.

For (2), Figure 2 (b) fixes the total communication to be $2^{30} \approx$ 1B `FP32` elements and varies the size per `All-Gather`, *i.e.*, smaller `AllGather` size means more `AllGather` invocations. Once the `AllGather` size decreases below 33M elements, the total communication time begins increasing rapidly.

Thus, to deliver highly efficient communications, FSDP organizes all parameters within one FSDP unit into a large `FlatParameter`, where the `FlatParameter` coalesces the communications of its individual parameters and also evenly shards them across ranks. More specifically, the `FlatParameter` is a 1D tensor constructed by concatenating p flattened original parameters and padding on the right to achieve a size divisible by the sharding factor. To shard the `FlatParameter`, FSDP divides it into equal-sized chunks, where the number of chunks equals the sharding factor, and assigns one chunk per rank. The `FlatParameter`'s gradient inherits the same unsharded and sharded shapes from the `FlatParameter`, and the `FlatParameter` and its gradient own the underlying storage of the original parameters and their gradients, respectively. Figure 3 depicts one example, where we use one FSDP unit to shard a 4×3 `nn.Linear` layer across 16 GPUs. In this case, every GPU only holds one element from the `FlatParameter` with the last rank holding the padded value.

This flatten-concat-chunk algorithm permits each original parameter to have arbitrary shape while minimizing the required padding (to be at most $F - 1$), reflecting its generality. Moreover, under this algorithm, the sharded and unsharded `FlatParameter` and its gradient have the exact data layout expected by `AllGather` and `ReduceScatter`, respectively. This enables calling the collectives without any additional copies for either the input or output tensors.

More formally, suppose for a model with Ψ number of elements, FSDP constructs N `FlatParameter`s with numels $\psi_1, \ldots, \psi_N$, where $\sum_{i=1}^{N} \psi = \Psi$. For sharding factor F, the peak parameter memory contribution is in $O(\sum_{i=1}^{N} \frac{\psi_i}{F} + \max_{i=1}^{N} \psi_i)$ because FSDP always keeps each local sharded `FlatParameter` with size $\frac{\psi_i}{F}$ in GPU memory and must materialize each unsharded `FlatParameter` with size ψ_i one by one during forward and backward. Since the first $\sum_{i=1}^{N} \psi_i = \Psi$ is fixed, the peak parameter memory contribution is determined by $\max_{i=1}^{N} \psi_i$. At the same time, the number of collectives per iteration is in $O(N)$. This evidences FSDP's memory-throughput trade-off: Finer-grained `FlatParameter` construction decreases peak memory but may decrease throughput by requiring more collectives. Users can control this trade-off by specifying how to wrap sub-modules into FSDP units.

3.2.2 *Hybrid Sharding.*

We refer to the strategy when the sharding factor is greater than 1 but less than W as *hybrid sharding*, as it combines both sharding and replication. For global world size W and sharding factor F, the parameters are sharded within each group $S_1, \ldots, S_{W/F}$ and are replicated within each complementary group $R_1, \ldots, R_F$, where each $S_i, R_j \subseteq \{1, \ldots, W\}$ gives the ranks in the sharded or replicated group, respectively.

For gradient reduction, the single reduce-scatter over all ranks becomes a reduce-scatter within each of the sharded groups followed by an all-reduce within each of the replicated groups to reduce the sharded gradients. The equivalence follows from the decomposition

$$\sum_{r=1}^{W} g_r = \sum_{i=1}^{W/F} \sum_{r \in S_i} g_r, \tag{1}$$

where g_r represents the gradient on rank r.

Hybrid sharding can take advantage of datacenter locality for accelerated training and can reduce cross host traffic to avoid as much contention in the oversubscribed environment as possible. At the same time, it provides a graduating trade-off between memory saving and throughput degradation, which is particularly helpful for models whose required memory footprint when trained with full replication is just slightly above the device capacity and do not want full sharding. Figure 4 shows one example.

Specifically, datacenters typically adopt a fat-tree network topology [16] with over-subscription, leading to abundant locality to exploit and a well-motivated reason to reduce cross-host traffic [17]. Hybrid sharding can provide a natural mechanism to map the device mesh into the datacenter layout to exploit such locality. For example, consider a cluster as a group of W accelerators grouped into hosts of of G accelerators each (where the communication among accelerators on the same host is much faster than the communication across hosts), we can set $F = \frac{W}{G}$ to limit the `AllGather` (and `ReduceScatter`) operations within the same host, while creating a replication group for accelerators with the same local rank across hosts. For an M-sized model, we can then compute the total cross-host traffic per GPU in the hybrid setup to be $2M\frac{W-1}{GW}$, a drastic reduction compared to full replication's $2M\frac{W-1}{W}$ and full sharding's $3M\frac{W-1}{W}$. Additionally, since the `AllReduce` collectives used in hybrid sharding operates at a smaller world size, they empirically achieve a better performance than invoking collectives at the global scale (in the case of full replication and full sharding), due to straggler effects and larger network interference.

Another important design motivation for hybrid sharding is the needs from medium-sized models. These models are large enough to cause out of memory issues when trained with full replication but are not large enough to fully utilize accelerator memory when used with full sharding, leading to *both* runtime overhead and memory waste. The hybrid sharding strategy creates a much richer memory-throughput trade-off space by simply adjusting F.

3.2.3 *Autograd.*

FSDP's `FlatParameter` must inter-operate with PyTorch's autograd engine to ensure (1) correct gradient propagation and (2) timely gradient reduction. For (1), recall that the `FlatParameter` and its gradient own the underlying storage of the original parameters and their gradients, respectively. To achieve this, before forward computation, FSDP sets the original parameters to be views into their unsharded `FlatParameter` using *autograd-visible* `torch.split()` and `torch.view()` calls. Then, the autograd engine naturally allocates the unsharded `FlatParameter` gradient and writes each original parameter's gradient to the appropriate offset as defined by `torch.split()`'s backward function. For (2), FSDP registers a gradient hook that only runs once the `FlatParameter`'s gradient is finalized. The hook represents the post-backward logic and includes the gradient reduction. Notably, FSDP's approach builds on top of PyTorch's autograd engine instead of hacking around it. As a result, FSDP automatically handles unconventional cases such as when not all parameters are used in the forward or when there are multiple forwards before a backward.

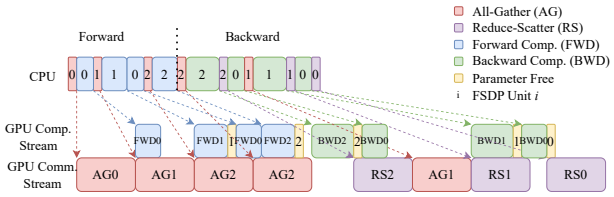


Figure 5: Overlap Communication and Computation

3.3 Communication Optimizations

The FSDP framework incorporates a range of native communication optimization techniques. This section unveils four major ones: overlapping, backward pre-fetching, forward pre-fetching, and accumulation.

3.3.1 *Overlapping Communication and Computation.*

The PyTorch `c10d` library has a `ProcessGroup` abstraction that represents a group of processes that can run collectives together. For the NCCL backend, the `ProcessGroupNCCL` implementation has an internal NCCL stream per device, where the separate internal stream is for asynchronous execution with the current stream, which is typically the default stream running computation. Those asynchronous collectives return `Work` objects, where calling `Work.wait()` blocks the CPU thread until the collective finishes. For general correctness, `ProcessGroupNCCL` synchronizes the internal stream with the current stream before running the collective. `DistributedDataParallel` leverages the async-collective-and-`wait()` approach to overlap the gradient `All-Reduce`s with backward computation. However, in contrast to DDP's backward where the `AllReduce` *proceeds* the computation with which to overlap, FSDP's forward issues the `AllGather` *following* the computation with which to overlap since in eager execution, FSDP cannot know which `FlatParameter` to `AllGather` next to reorder it before the computation. This difference in kernel-issue order makes following the async-collective-and-`wait()` approach infeasible for FSDP. Namely, since `ProcessGroupNCCL` synchronizes with the current (default) stream, the `All-Gather` will not run until the computation with which to overlap finishes. To address this, FSDP uses a separate CUDA stream to issue the `AllGather`s, bypassing the false dependency on preceding computation in the default stream and allowing each `AllGather` to overlap. As a result, FSDP's collective synchronization operates on streams, not simply `Work` objects. Figure 5 illustrates one example. Note that the backward pass excludes the `AG0` `All-Gather` because FSDP intentionally keeps the outermost FSDP unit's parameters in memory to avoid redundantly freeing at the end of forward and then re-`All-Gather`ing to begin backward.

3.3.2 *Backward Prefetching.*

FSDP enforces a single CUDA device per rank and uses a single process group for both `AllGather` and `ReduceScatter`, which means that its collectives run sequentially in the process group's internal NCCL stream. In the backward pass, FSDP issues the `ReduceScatter` for the current `FlatParameter` and then the `AllGather` for the next `FlatParameter`. Hence, the single NCCL stream forces the `ReduceScatter`

to block the next AllGather, which in turn blocks the next gradient computation and may become exposed on the critical path.

To avoid two consecutive exposed communication calls in the backward pass, FSDP *backward prefetching* issues the next AllGather before the current ReduceScatter. However, as mentioned before, a challenge for eager execution is knowing which FlatParameter to AllGather next. FSDP resolved this challenge by recording the reverse forward execution order of modules as the proxy of their backward execution order. Moreover, the forward order is freshly recorded each iteration, meaning that the *backward prefetching* is compatible with dynamism across iterations.

3.3.3 *Forward Prefetching.*

For some workloads with relatively slow CPU execution, the CPU thread may not be able to issue the next forward AllGather early enough to efficiently fill the NCCL stream. If the model follows a static computational graph across iterations, then FSDP can assume the forward execution order of modules from the previous iteration and prefetch the next AllGather explicitly in the forward pass. This *forward prefetching* issues the next AllGather before forward computation of current FSDP unit.

3.3.4 *Gradient Accumulation.*

FSDP offers two variations of gradient accumulation: with and without communication. With communication, FSDP still reduces gradients across ranks, and each rank saves the sharded gradients. Simply running multiple iterations without clearing gradients achieves this. Without communication, FSDP does not reduce gradients across ranks, and each rank saves the unsharded gradients. This latter variation trades off increased memory usage with decreased communication, which can increase end-to-end throughput.

3.4 Memory Management

PyTorch uses a CUDA caching allocator as a middle layer to serve GPU allocation and free requests for PyTorch programs. In order to effectively manage memory, FSDP uses a *rate limiter* to take into account the memory impact of the caching allocator on programs that use several CUDA streams and run fast CPU threads.

3.4.1 *How Does PyTorch Caching Allocator Affect Memory.*

The caching allocator avoids frequent calls to cudaMalloc and cudaFree, where the latter incurs a costly device synchronization. Specifically, the caching allocator requests CUDA memory blocks and internally determines how to split and reuse the blocks without returning them to CUDA with the goal being to reach a steady state without further calls to cudaMalloc and cudaFree.

The caching allocator runs from the CPU thread, meaning that it must decide which caching allocator block to use for an allocation when the *CPU thread* processes the allocation request. It cannot wait until the GPU kernel needing the allocation actually runs, which may be much later.

For a single stream, the caching allocator can directly reuse memory blocks by the stream's sequential ordering semantics. However, for separate producer and consumer streams, there are no inter-stream ordering guarantees, and the caching allocator cannot be certain that a block is safe to reuse until the last *GPU kernel* depending on that memory finishes running. Hence, if the CPU thread runs far ahead of the GPU execution, then the caching allocator cannot reuse blocks for the producer stream with pending GPU kernels from the consumer stream.

Furthermore, caching allocator blocks are allocated *per stream* and cannot be reused for a different stream, this over-allocates blocks to the producer stream that could otherwise be used for the consumer stream (*e.g.* for activations). The GPU itself may have enough memory to serve a new allocation in the consumer stream, but the overallocation to the producer stream may lead to the caching allocator failing to serve it. This forces a blocking sequence of cudaFrees to reset the caching allocator memory state called a cudaMalloc retry that greatly degrades training throughput.

3.4.2 *Rate Limiter.*

FSDP allocates the AllGather destination tensor representing the unsharded FlatParameter in a producer stream, and the forward and backward computations using the AllGathered parameters run in a consumer stream (typically the default stream). For a fast CPU thread, there may be pending GPU computation kernels when the caching allocator must serve the next AllGather, leading to no block reuse. Even after the blocks are not active in the AllGather producer stream, these reserved blocks can not serve default computation stream's allocation requests, and thus may force blocking cudaFrees and cudaMallocs.

FSDP offers a *rate limiter* that intentionally blocks the CPU thread to ensure proper caching allocator block reuse. It allows at most two inflight AllGathers, which is the minimum amount to still achieve communication and computation overlap.

4 IMPLEMENTATION

This section delves into the intricacies of FSDP implementation, which although do not alter the FSDP core algorithm, are crucial to understand before adopting FSDP.

Users can access FSDP through two APIs, FullyShardedDataParallel model wrapper and fully_shard module annotator. The former wraps the entire model and replaces sub-modules with corresponding FSDP units. In contrast, the latter installs FSDP logic as nn.Module forward and backward hooks, preserving both model structures and parameter fully-qualified names.

4.1 Initialization

Section 3.2.1 described FSDP's solution to efficiently initialize large models, which works well when sub-module initializations are self-contained. In a rare situation where one sub-module's initialization depends on a parameter from the different sub-module, the on-demand materialization and record-replay approach might break if the parameter belongs to a different FSDP unit, because the unsharded version of that parameter could have been discarded to reduce memory footprint. Therefore, besides the advanced deferred initialization, FSDP offers two more options:

- **Initialize unsharded model on GPU**. The memory requirement for model initialization may be smaller than that for training since training also involves gradients, activations, and optimizer states. Consequently, if the training

step cannot be performed on a single GPU device, users might still be able to initialize the entire model on a GPU and pass it to FSDP. Then, optimizers should be instantiated after FSDP shards the model, to reduce the memory footprint and align with the sharded gradients produced by FSDP.

- **Initialize unsharded model on CPU**. If the size of the unsharded model surpasses the capacity of GPU memory and can only be accommodated in CPU memory, it becomes impracticable to move the unsharded model entirely to the GPU before handing it over to FSDP for parameter sharding. To overcome this challenge, FSDP adopts a streaming approach, where the model is migrated to the GPU unit by unit. Upon arrival to the GPU, the parameters of each unit are immediately sharded, which in turn reduces the memory overhead before processing the next unit. This approach remains viable even when there are cross-submodule dependencies during initialization, given that all parameters of the entire unsharded model are present in the CPU memory.

Note that both approaches above are subject to their own limitations. The first method entails the entire model fitting within a single GPU device and thus becomes infeasible for larger models. The second method, on the other hand, can handle larger models since the CPU has considerably larger memory. However, this approach may experience substantial slowdowns in comparison to deferred initialization due to the limited memory bandwidth and parallelization capabilities of the CPU. In light of these observations, users may still prefer deferred initialization, even when dealing with models of the size range encompassed by the previous two methods.

To delimit the scope of each FSDP unit, users may choose to employ the `FullyShardedDataParallel` wrapper by intrusively applying it to sub-modules in model source code, or alternatively, provide a custom function to the `auto_wrap_policy` argument upon instantiation. Selecting the optimal wrapping approach typically requires some experiments and measurements.

4.2 Flat Parameters

The `FlatParameter` class inherits from `nn.Parameter` and behaves like an `nn.Parameter`. FSDP implements an accompanying `FlatParamHandle` class that is responsible for managing individual `FlatParameter` instances. The frontend, either `FullyShardedDataParallel` or `fully_shard`, interfaces with the `FlatParameter`s only through `FlatParamHandle`.

One `FlatParameter` accommodates storage for all parameter tensors within one FSDP unit. The boundary of the FSDP unit controls the timing for `AllGather` and `ReduceScatter`, which has a direct impact on overall FSDP performance. In the ideal case, FSDP unit boundaries should align with model execution order.

FSDP has access to the model's static `nn.Module` structure at construction time. Fortunately, although this structure does not guarantee to faithfully represent model execution order, model authors conventionally translate layers and broader blocks to nested `nn.Module` definitions that may naturally have the desired parameter locality. FSDP can leverage that structure to choose the `FlatParameter` construction. Indeed, FSDP supports annotating `nn.Module`s and follows a simple rule: All parameters in the annotated `nn.Module` are assigned to one `FlatParameter`, excluding those parameters already assigned. This rule lends itself naturally to nested annotation, where blocks are annotated, forming well-sized `FlatParameter`s, and any residual parameters are assigned to their parent.

Another approach we explored is using the execution order and reconstructing `FlatParameter`s dynamically. This approach starts with an initial small `FlatParameter` construction, runs a possibly inefficient first iteration while observing the execution order, and reconstructs the `FlatParameter`s by coalescing the existing small `FlatParameter`s according to the observed order.

4.3 Runtime

FSDP augments a local model instance by incorporating communication operations to reduce gradients and gather parameters. Timely initiation of these operations is of paramount importance for ensuring both correctness and efficiency. Starting communication too soon would cause the parameters or gradients with pending updates to be consumed, while initiating communication too late would result in wasting network bandwidth and delay in subsequent computations.

To insert communication-related code to the model forward pass, the `FullyShardedDataParallel` `nn.Module` wrapper overrides `nn.Module`'s `forward()` method to install pre-forward and post-forward logic, whereas the functional `fully_shard` implements them by registering `nn.Module` hooks through methods such as `register_forward_pre_hook()` and `register_forward_hook()`. It is more challenging to capture appropriate signals from the backward pass, as PyTorch automatically and transparently handles the backward pass. Fortunately, the autograd engine exposes a variety of hooks that enable the installation of custom logic with precise granularity.

- **Hooks on `Tensor`** through `register_hook()` allows to run custom function when the gradient of a `Tensor` is generated. This can help anchor FSDP logic to an activation's gradient computation in the backward pass. FSDP registers this type of hook to the forward output tensor of every FSDP unit to insert communications before backward pass enters that FSDP unit.
- **Hooks on `backward()`** through `queue_callback()` run right before exiting the current autograd `GraphTask`, which is usually the end of overall backward pass. FSDP relies on this hook to wait for pending communications so that the subsequent optimizer step will not consume gradients too early.
- **Hooks on `AccumulateGrad`** autograd function fires when the gradient of a parameter has finished accumulation in the current backward pass. FSDP attaches this type of hook to each `FlatParameter`'s `AccumulateGrad` function to immediately launch `ReduceScatter` when gradients are ready. Note that the `Tensor` hook mentioned above can potentially achieve the same behavior, but might incur unnecessary delay as it needs to wait for gradient computations for input activations as well.

The aforementioned methodologies collectively integrate the FSDP algorithm with the PyTorch `nn.Module` and autograd engine in a non-intrusive and efficient manner.

4.4 Native Mixed Precision

FSDP offers a versatile native mixed precision mechanism. In terms of parameter management, it adheres to the standard mixed precision technique, which maintains both low and full precision copies of parameters [18]. Forward and backward computation use the low precision, and the optimizer step uses full precision. FSDP permits user-specified precisions for parameters, gradient reduction, and non-trainable buffers, each independently if desired.

For Ψ number of parameter elements (`torch.numel`), K_{low} bytes per low precision element, and K_{full} bytes per full precision element, this approach to mixed precision normally increases the memory overhead from $K_{\text{full}}\Psi$ to $(K_{\text{low}} + K_{\text{full}})\Psi$ due to maintaining both precision copies. However, FSDP can sidestep the problem given our design to always keep each local sharded `FlatParameter` in GPU memory and only dynamically allocate the unsharded `FlatParameter`. For N `FlatParameter`s with numels given by $\psi_1, \ldots, \psi_N$, the parameter peak memory contribution for FSDP actually *decreases* from $\frac{K_{\text{full}}}{F} \sum_{i=1}^{N} \psi_i + K_{\text{full}} \max_{i=1}^{N} \psi_i$ to $\frac{K_{\text{full}}}{F} \sum_{i=1}^{N} \psi_i + K_{\text{low}} \max_{i=1}^{N} \psi_i$ bytes. In other words, FSDP directly reduces the second $K_{\text{full}} \max_{i=1}^{N} \psi_i$ term to $K_{\text{low}} \max_{i=1}^{N} \psi_i$.

In contrast to `torch.amp.autocast` that performs just-in-time casts at the operator level, FSDP's native mixed precision only incurs a full-to-low-precision cast per `FlatParameter` in its pre-forward and, if resharding after forward, its pre-backward. Moreover, FSDP's mixed precision permits running all collectives in the low precision, which saves communication volume.

Users most commonly choose `FP16` or `BF16` as the low precision and `FP32` as the full precision. `FP16`'s smaller dynamic range compared that of `FP32` exposes `FP16` to greater risk of numeric underflow and overflow. The standard solution includes a gradient scaler [1] that scales gradients to a safe magnitude. However, since FSDP shards gradients across ranks, a normal local gradient scaler implementation breaks mathematical equivalence, and instead, FSDP provides its own sharded gradient scaler.

5 EVALUATION

We conducted an empirical evaluation of FSDP on large language models and recommendation system models and compared the results with those of DDP. Experiment specifications are described in Section 5.1. Then, we organize experiments into three categories. Section 5.2 focuses on how well FSDP handles different sizes of models. Then, Section 5.3 discusses the impact of throttling communications. Finally, Section 5.4 demonstrate FSDP's ability to scale to gigantic models.

5.1 Experiment Setup

In these experiments, we conducted evaluations on the HuggingFace T5-11B transformer [26], minGPT-175B transformer [3], and DHEN recommendation model [33]. The recommendation models consist of 768B sparse parameters and 550M dense parameters, the sparse parameter tensors were sharded using the first approach mentioned in Section 2.3, which communicates activations instead of parameters, while the dense parameters were trained using FSDP on 8 to 512 A100 80GB GPUs interconnected by a 2Tb/s RoCE network. The objective was to assess the capability and scalability of FSDP in training large-scale models. Additionally, we employed T5-611M, T5-2B and T5-11B transformers to evaluate the performance of various sharding strategies, communication efficiency of prefetching, and communication throttling using rate limiter. Metrics employed in these experiments included TFLOPS per GPU, latency per batch, peak memory allocated, peak memory active, and peak memory reserved.

5.2 Model Scale

In this section, we investigate the performance of FSDP when dealing with models of different sizes, spanning from 611M to 175B, and make a comparison with DDP [14].

The experimental results on T5 models are displayed in Figure 6 (a). The performance of FSDP and DDP is similar when evaluating 611M and 2.28B models. However, DDP encounters an out-of-memory error when attempting to wrap models larger than 2.28B. In contrast, FSDP can effortlessly accommodate the 11B model and achieve significantly higher TFLOPS by turning on `BF16`. These experiments illustrate that practitioners can utilize FSDP for both small and large models, and seamlessly transition across different model configurations.

Then, we conduct additional experiments to measure the acceleration attained through backward pre-fetching. This time we use a larger GPT-175B model, where communication overhead is more prominent. Results are presented in Figure 6 (b), where pre-fetching leads to approximately 18% speedup, and this TFLOPS gain persists across different GPU cluster sizes. Therefore, for subsequent experiments, we always turn-on backward pre-fetching.

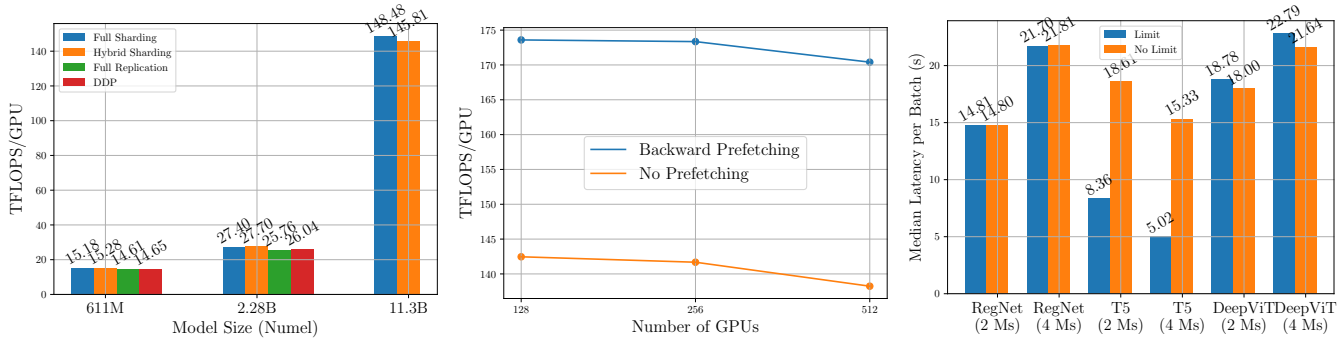
5.3 Throttle Communications

In the subsequent analysis, we investigate the implications of throttling FSDP communications. As expounded in Section 3.4, launching `AllGather` too aggressively can lead to unnecessarily high memory footprint, as the CPU thread needs to allocate CUDA memory blocks when the communication kernel is added into the CUDA stream. This predicament may sometimes result in significant performance problems when the CPU thread runs too fast in comparison to CUDA streams. To gauge its efficacy in varying scenarios, we apply rate limiting to three different types of models and applied the maximum feasible batch size in each experiment.

- **RegNet** [29]: model size 9B, and batch size 48 for 2 nodes and 72 for 4 nodes.
- **T5** [26]: model size 11B, and batch size 2.
- **DeepViT** [36]: model size 8B, and batch size 105 for 2 nodes and 120 for 4 nodes.

Experiment results are plotted in Figure 6 (c). One notable observation is that the rate limiter's effectiveness is not consistent, as it does not attain any speedups in the RegNet experiments, and even impedes the DeepViT ones. This behavior is expected since throttling the communications can only boost training if the fast CPU thread aggressively allocates GPU memory blocks and causes defragmentations. If it is difficult to identify with certainty from latency measurements or profiled traces, CUDA `malloc` retry can serve as a helpful indicator, which can be obtained from the `num_alloc_retries` key in the `torch.cuda.memory_stats()` dictionary.

The experiments conducted with T5 models have demonstrated that the rate limiter technique can greatly benefit training efficiency,



(a) Model Scale

(b) GPT-175B Backward Prefetch

(c) Rate Limiter (Ms = Machines)

Figure 6: Model Scale and Training Efficiency

yielding up to 5X speedups. However, for DeepViT models, introducing communication throttling can result in an additional 5% overhead. This is due to the fact that delaying the `AllGather` communication can potentially block subsequent model computations that rely on the `AllGather`ed parameters, especially in cases where communication is the dominant factor. Therefore, before enabling rate limiting, practioners should verify whether defragmentation has taken place during training.

5.4 Efficient Training for Large Models

To evaluate capability of using FSDP for large models, we ran three types of models using Full Sharding with prefetching and rate limiter turned on. Activation checkpointing and `BF16` mixed precision are also applied in these experiments. Adam optimizer is used to reflect a production workload setup and to incur the costly two optimizer states per parameter.

- **DHEN large recommendation model** [33]: model size - 768B sparse parameters and 550M dense parameters, and batch size 1024.
- **minGPT transformer** [10]: model size 175B, vocab size 50000, block size 2048, batch size 1 and 2 for 128, 192, 256, 384 and 512 GPUs.
- **HuggingFace T5 transformer** [26]: model size 11B, sequence length 512, batch size 8 and 16 for 8, 16, 32, 64, 128, 256, 512 GPUs.

In the DHEN experiments, we further combine sharding strategies with two different configurations:

- **RAF:** reshard-after-forward frees `AllGather`ed shards from other GPUs after forward pass and unshards them again before backward computation. This reduces peak memory consumption at the cost of higher communication overhead.
- **NRAF:** no-reshard-after-forward is the opposite where the unsharded model parameters stay in GPU memory after forward pass until backward computations finish, which trades higher memory footprint for lower communication overhead.

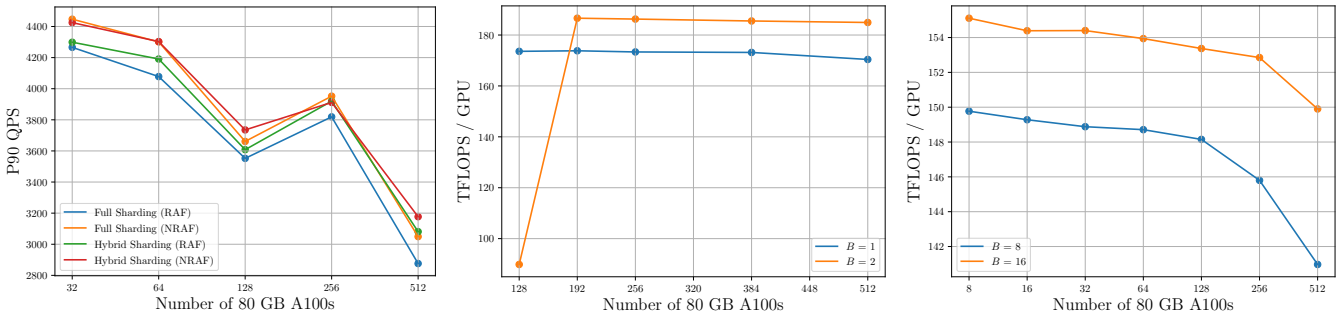
The experimental results in Figure 7 (a) and Figure 8 (a) indicate that FSDP is capable of accommodating DHEN models on a large GPU cluster. It was observed that Full Sharding with RAF yields the smallest memory footprint but with a corresponding trade-off of reduced QPS. Conversely, Hybrid Sharding with NRAF demonstrated the opposite behavior, as it has employs both a smaller sharding group and skips one reshard. When adding more GPUs to in the cluster, the peak memory usage consistently decreases as a result of a decrease in the size of each rank's model shard.

With the 175B model, the experiments achieved more than 173 and 186 TFLOPS per GPU with batch size equal to 1 and 2 respectively as shown in Figure 7 (b). This is equivalent to approximately 55% and 60% of GPU hardware utilization, given that the A100's peak is 312 TFLOPS using the `BF16` tensor core. Furthermore, the model demonstrated linear scalability from 128 GPUs to 512 GPUs, in terms of TFLOPS, which affirms the efficacy of FSDP in handling large models with expensive computations or high-speed network interconnections. Notably, with 128 GPUs, setting the batch size to 2 resulted in a considerably lower per-GPU TFLOPs in comparison to other scenarios. This was due to CUDA memory defragmentation during the backward pass. The backward pass contributed 85.56% of the iteration latency for the 128 GPU batch size equals 2 case, while a normal backward pass only accounted for about 67% in these experiments. Using 128 GPUs is more likely to trigger defragmentation, as each GPU needs to accommodate a larger model shard. Figure 8 confirms this explanation, where the PyTorch CUDA caching allocator depletes all 80GB of the CUDA memory as shown on the top left corner.

Finally, for T5-11B models as shown in Figure 8 (c), all experiments are executed comfortably below GPU memory capacity, where defragmentations are unlikely to happen. Nevertheless, as the number of GPUs increases from 8 to 512, a 7% regression in per-GPU TFLOPS is still evident as illustrated in Figure 7 (c). This suggests that communications begin to outweigh computations on large clusters, and a near-perfect overlap between communication and computation is no longer attainable.

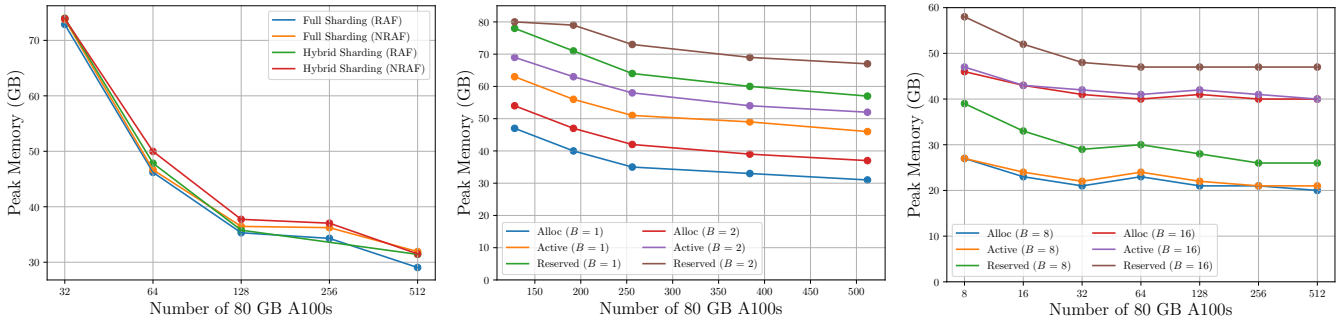
6 RELATED WORK

The DDP [14] model wrapper, which is based on the model replication design, was an initial distributed training feature introduced in PyTorch [24]. Although it can handle large datasets, it cannot accommodate the ever-increasing model sizes that are now prevalent in the field.



(a) DHEN QPS (b) GPT-175B TFLOPS (c) T5-11B TFLOPS

Figure 7: Training Throughput: To conform with DHEN convention, we use sample/ GPU/second (QPS) for DHEN.



(a) DHEN (b) GPT-175B (c) T5-11B

Figure 8: Memory Footprint

ZeRO [27, 28] and cross-replica sharding [30] inspired the FSDP design, but FSDP is intrinsically different. Prior work employs model partitioning or per-parameter sharding to distribute parameter tensors, and rely on `Broadcast` and `Gather` collective communication primitives to synchronize values. Although this design can achieve the same functionality, it could lead to uneven workload distribution across GPU devices, which hampers the efficiency of synchronized distributed training. Additionally, since this approach modifies the internals of the machine learning framework, such as tensor storage and memory management, it might no longer work when the internal implementation is updated or new features are introduced. Therefore, a native solution that is co-designed with the core components of the framework would provide a more robust and consistent user experience.

MiCS [34] and FSDP differ in gradient communication strategies. MiCS uses a global `AllReduce` followed by sharding within each partition group, whereas FSDP employs `AllGather` and `ReduceScatter`. As a result, each rank in MiCS must hold the entire model gradients, leading to higher memory usage than FSDP's approach of sharding a single layer. While both MiCS and FSDP use a hybrid communication strategy to improve efficiency at scale, FSDP's approach schedules `AllGather` within a flexibly-sized sharded group, potentially resulting in lower runtime latency than the two-hop `AllGather` utilized by MiCS. This reduced latency is crucial as the `AllGather` operation is critical to execution, and limiting the world size and participants of `AllGather` to accelerators within a group with good locality can result in lower latency and higher throughput.

Pipeline parallelism [5, 8] involves partitioning model parameters and their activations across multiple devices through the division of models into pipeline stages. However, pipeline parallelism requires model changes and meticulous tuning for microbatch sizes, number of stages and partitions, as well as intricate scheduling procedures to optimize performance by squeezing out bubbles.

Additionally, specific attention is given to high profile architectures such as transformers. For example, sequence parallelism [13] reduces activation memory in conjunction with tensor parallelism; Pipetransformer [6] designed a dynamic 2D parallelism that allows changing the dimensions of pipeline and data parallelism on the fly, depending on learning signals. These methods are highly effective but can be difficult to generalize as they either rely on the specific implementation or the model's layered structure.

Many existing solutions combine data parallelism with other parallelisms to achieve speedup. For example, Megatron [21] demonstrated highly efficient deep transformer training on large clusters using 3D (data, tensor and pipeline) parallelism. Further, compiler-based techniques such as Alpa [35], GSPMD [31], and FlexFlow [9] leverage profiling, performance modeling, user annotations and search to find the best configuration across the parallelism space of

data, tensor and pipeline for a given cluster. In all cases, FSDP provides the benefit of being a drop-in replacement for data parallelism that reduces data redundancy along the data parallel axis.

Orthogonal memory-saving techniques include gradient compression [2], mixed-precision training [7], tensor rematerialization [12] and CPU-offloading [4], but they could have implications on model accuracy and incur overhead in (un)compression, quantization, recomputation, and host-to-device copies, respectively.

7 DISCUSSION

This section discusses how FSDP can be combined with other parallelism paradigms and known limitations when adopting FSDP.

7.1 FSDP Interoperability

Further increasing scalability and efficiency of distributed training requires combining FSDP with other paradigms. This section briefly highlights how the FSDP design enables mixing and matching with other types of parallelisms.

7.1.1 Pipeline Parallelism.

Pipeline parallel can be functionally integrated with FSDP by employing FSDP to wrap each individual pipeline stage. However, as pipeline parallel divides input mini-batches into smaller micro-batches, the default full sharding strategy in FSDP would have to unshard model parameters for every micro-batch. Consequently, combining these approaches with default FSDP configurations may lead to significant communication overhead. Fortunately, FSDP offers alternative sharding strategies that can keep parameters unsharded after the forward pass, avoiding unnecessary `AllGather` communications per micro-batch. Admittedly, this requires storing parameters of an entire pipeline stage on the GPU device, but FSDP can still reduce memory usage as it still shards gradients and optimizer states.

7.1.2 Tensor Parallelism.

In contrast to FSDP, tensor parallel keeps parameters sharded during computation, which is necessary if any sub-module is too large to fit in GPU memory. Presently, PyTorch provides a prototype feature called `parallelize_module` that can be combined with FSDP to construct 2D parallelism. It works by organizing devices into a 2D mesh where PyTorch's distributed tensor `DTensor` manages tensor parallelism on one dimension and FSDP applies sharded data parallelism on the other dimension. These two dimensions communicate activations and parameters, respectively. We usually keep the tensor-parallel communications, which block subsequent computation, intra-node to leverage the higher network bandwidth, and allow the FSDP communications operate on the other mesh dimension inter-node.

7.2 Limitations

During our work with production and research applications, we have encountered certain limitations associated with FSDP. This section aims to discuss two tricky caveats that are not readily apparent and pose significant challenges when it comes to troubleshooting.

7.2.1 Mathematical Equivalence.

FSDP cannot ensure that it always achieves the same mathematical equivalence as local training, especially with respect to the optimizer computation. This stems from the fact that the optimizer step operates on the sharded parameters, whose data layout is a function of FSDP's `FlatParameter` sharding algorithm that does not respect individual parameter boundaries. As a result, any optimizer computation that depends on an original parameter's unsharded value (*e.g.* vector norm), its tensor structure (*e.g.* approximate second-order optimizers), or require global states over all parameters will become invalid. Addressing this requires uneven sharding, padding, or extra communication, all of which hurt performance. Co-designing such optimizer computations with sharding is an open research question.

7.2.2 Shared Parameters.

For shared parameters, FSDP must ensure to not flatten them into multiple `FlatParameter`s and to ensure that they are unsharded properly when needed for all usages. If handled incorrectly, PyTorch may raise an error regarding missing tensor storage or size mismatch, which can happen when an FSDP unit attempts to use a shared parameter that has already been resharded by a preceding FSDP unit. The current recommendation is to construct FSDP units such that the shared parameter belongs to the lowest-common-ancestor unit to ensure that the shared parameter is unsharded throughout all usages. This may require some inspection of the model structure to do correctly and may undesirably keep the `FlatParameter` unsharded for a large interval, so we are investigating approaches to improve shared parameter handling.

8 CONCLUSION

This manuscript elucidates the underlying rationale, design philosophy, and implementation of `FullyShardedDataParallel` as of PyTorch 2.0 release. FSDP attains usability and efficiency through a set of advanced techniques, including deferred initialization, flexible sharding strategies, communication overlapping and prefetching, and rate limiting communication collectives. All of these techniques are closely co-designed with other key PyTorch components to ensure the solution is sound and robust. Evaluations show that FSDP can facilitate large language and recommendation models with near linear scalability.

ACKNOWLEDGMENTS

We are grateful to the PyTorch community and PyTorch FSDP users for their feedback and contributions.

REFERENCES

[1] 2023. `torch.amp` Gradient Scaling. https://pytorch.org/docs/2.0/amp.html#gradient-scaling.

[2] Youhui Bai, Cheng Li, Quan Zhou, Jun Yi, Ping Gong, Feng Yan, Ruichuan Chen, and Yinlong Xu. 2021. Gradient compression supercharged high-performance data parallel dnn training. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 359–375.

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[4] Jiarui Fang, Zilin Zhu, Shenggui Li, Hui Su, Yang Yu, Jie Zhou, and Yang You. 2022. Parallel Training of Pre-Trained Models via Chunk-Based Dynamic Memory Management. *IEEE Transactions on Parallel and Distributed Systems* 34, 1 (2022), 304–315.

[5] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. 2018. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377* (2018).

[6] Chaoyang He, Shen Li, Mahdi Soltanolkotabi, and Salman Avestimehr. 2021. Pipetransformer: Automated elastic pipelining for distributed training of transformers. *arXiv preprint arXiv:2102.03161* (2021).

[7] Xin He, Jianhua Sun, Hao Chen, and Dong Li. 2022. Campo: Cost-Aware Performance Optimization for Mixed-Precision Neural Network Training. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 505–518. https://www.usenix.org/conference/atc22/presentation/he

[8] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems* 32 (2019).

[9] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2018. Beyond Data and Model Parallelism for Deep Neural Networks. https://doi.org/10.48550/ARXIV.1807.05358

[10] Andrej Karpathy. 2020. MinGPT Transformer model. https://github.com/karpathy/minGPT.

[11] Chiheon Kim, Heungsub Lee, Myungryong Jeong, Woonhyuk Baek, Boogeon Yoon, Ildoo Kim, Sungbin Lim, and Sungwoong Kim. 2020. torchgpipe: On-the-fly pipeline parallelism for training giant models. *arXiv preprint arXiv:2004.09910* (2020).

[12] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. 2020. Dynamic Tensor Rematerialization. https://doi.org/10.48550/ARXIV.2006.09616

[13] Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2022. Reducing activation recomputation in large transformer models. *arXiv preprint arXiv:2205.05198* (2022).

[14] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. 2020. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704* (2020).

[15] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. 2021. Terapipe: Token-level pipeline parallelism for training large-scale language models. In *International Conference on Machine Learning*. PMLR, 6543–6552.

[16] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. 2017. Incbricks: Toward in-network computation with an in-network cache. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. 795–809.

[17] Liang Luo, Peter West, Jacob Nelson, Arvind Krishnamurthy, and Luis Ceze. 2020. Plink: Discovering and exploiting locality for accelerated distributed training on the public cloud. *Proceedings of Machine Learning and Systems* 2 (2020), 82–97.

[18] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. 2017. Mixed Precision Training. https://doi.org/10.48550/ARXIV.1710.03740

[19] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, et al. 2021. High-performance, distributed training of large-scale deep learning recommendation models. *arXiv preprint arXiv:2104.05158* (2021).

[20] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 1–15.

[21] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. 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*. 1–15.

[22] NVIDIA. 2023. The NVIDIA Collective Communication Library (NCCL). https://developer.nvidia.com/nccl.

[23] OpenAI. 2023. ChatGPT. https://chat.openai.com/.

[24] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 8024–8035. http://papers.nips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[25] Team PyTorch. 2023. DISTRIBUTED RPC FRAMEWORK. https://pytorch.org/docs/stable/rpc.html.

[26] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research* 21, 1 (2020), 5485–5551.

[27] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[28] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. ZeRO-Offload: Democratizing Billion-Scale Model Training.. In *USENIX Annual Technical Conference*. 551–564.

[29] Nick Schneider, Florian Piewak, Christoph Stiller, and Uwe Franke. 2017. RegNet: Multimodal sensor registration using deep neural networks. In *2017 IEEE intelligent vehicles symposium (IV)*. IEEE, 1803–1810.

[30] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Hongjun Choi, Blake Hechtman, and Shibo Wang. 2020. Automatic cross-replica sharding of weight update in data-parallel training. *arXiv preprint arXiv:2004.13336* (2020).

[31] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, et al. 2021. GSPMD: general and scalable parallelization for ML computation graphs. *arXiv preprint arXiv:2105.04663* (2021).

[32] Jinhui Yuan, Xinqi Li, Cheng Cheng, Juncheng Liu, Ran Guo, Shenghang Cai, Chi Yao, Fei Yang, Xiaodong Yi, Chuan Wu, et al. 2021. Oneflow: Redesign the distributed deep learning framework from scratch. *arXiv preprint arXiv:2110.15032* (2021).

[33] Buyun Zhang, Liang Luo, Xi Liu, Jay Li, Zeliang Chen, Weilin Zhang, Xiaohan Wei, Yuchen Hao, Michael Tsang, Wenjun Wang, Yang Liu, Huayu Li, Yasmine Badr, Jongsoo Park, Jiyan Yang, Dheevatsa Mudigere, and Ellie Wen. 2022. DHEN: A Deep and Hierarchical Ensemble Network for Large-Scale Click-Through Rate Prediction. https://doi.org/10.48550/ARXIV.2203.11014

[34] Zhen Zhang, Shuai Zheng, Yida Wang, Justin Chiu, George Karypis, Trishul Chilimbi, Mu Li, and Xin Jin. 2022. MiCS: near-linear scaling for training gigantic model on public cloud. *arXiv preprint arXiv:2205.00119* (2022).

[35] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. 2022. Alpa: Automating Inter-and {Intra-Operator} Parallelism for Distributed Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 559–578.

[36] Daquan Zhou, Bingyi Kang, Xiaojie Jin, Linjie Yang, Xiaochen Lian, Zihang Jiang, Qibin Hou, and Jiashi Feng. 2021. Deepvit: Towards deeper vision transformer. *arXiv preprint arXiv:2103.11886* (2021).