# PyTorch Fully Sharded Data Parallel (FSDP)

Efficient Training of Large Neural Networks with High Scalability

Authors - Yanli Zhao Meta AI yanlizhao@meta.com,
Andrew Gu Meta AI andgu@meta.com,
Rohan Varma Meta AI rvarm1@meta.com

Presented by - Vrushali Harane

Credits : Skywork.ai

# Introduction and Motivation

## Unprecedented Scale

- GPT-3: **175 billion** parameters
- Recommendation models: **exceeding 1 trillion** parameters
- Performance gains come at enormous computational cost

## Accessibility Challenge

- Technical barriers limit development to select users
- Specialized hardware and expertise required
- Creates an uneven playing field in AI innovation

## Need for Distributed Training Solutions


### The Core Problem

Current distributed training methods require models to fit on a single GPU, creating a barrier for large model development. FSDP aims to democratize access to these powerful technologies by enabling efficient training of models that would otherwise exceed memory capacity.

# Existing Parallelism Techniques

## Pipeline Parallelism

Stage 1 — Stage 2 — Stage 3

- Partitions model into stages across devices
- Communicates activations across stage boundaries

⚠ Requires model-specific integration

## Tensor Parallelism

| shard 1 | shard 2 | shard 3 |
|---|---|---|
| W1 | W2 | W3 |

- Shards computation inside a layer across devices
- Computes on local shards, communicates as needed using *AllReduce/AllGather*

⚠ Tightly coupled to model architecture

## Zero-Redundancy Parallelism

Rank 1 Rank 2 Rank 3

On-demand parameter communication

- Shards parameters, gradients and optimizers across devices
- Communicates parameters on-demand

⚠ Vulnerable to framework changes

## Common Drawbacks

### Architecture-Specific Integration

Methods tightly coupled with specific model architectures, limiting general applicability

### Vulnerability to Framework Changes

Built upon rapidly evolving internal interfaces, making them susceptible to changes

# Background - Evolution of PyTorch Distributed Training

| Initial DDP | Pipeline Parallelism | Tensor RPC | FSDP |
| --- | --- | --- | --- |
| 2017 | 2019 | 2020 | 2021 |

## Model Replication

Foundation using **DistributedDataParallel (DDP)**

- ✓ Complete model replicas on each device
- ✓ Synchronizes gradients via AllReduce
- ✗ Requires all model params fit in single GPU
- ✗ Out-of-memory errors for large models

## Model Partitioning

Advanced techniques for larger models

- ✓ Divides model into smaller components
- ✓ Pipeline parallelism for sequential stages
- ✓ Tensor RPC for remote computations
- ⚠ Code modifications required

## Model Sharding

Efficient memory utilization approach

- ✓ Parameters distributed across ranks
- ✓ On-demand communication technique
- ✓ Each device holds only parameter shards
- ★ FSDP adopts this approach

💡 Evolution shows growing complexity to address memory limitations and enable larger model training

# Key Challenges Addressed by FSDP

## User Experience

Traditional methods require complete model replication

- Solves initialization hurdle with **deferred initialization**
- Creates models on dummy device, initializes unit-by-unit on GPU
- Enables efficient memory usage during initialization

## Hardware Heterogeneity

Modern GPU clusters have varying bandwidth interconnects

- Configurable sharding strategies for diverse hardware
- Optimizes communication patterns for varying bandwidth
- Adapts to hierarchical interconnects (high-bandwidth within machine)

## Resource Utilization

Minimizing downtime is crucial for GPU utilization

- Operation reordering to maximize overlap
- Parameter prefetching to bridge communication gaps
- "Squeezes out" idle times ("bubbles") during execution

## Memory Planning

Efficient memory management is paramount

- Optimizes memory usage with prudential allocation
- Restricts blocks allocated for in-flight unsharded parameters
- Suspends CPU execution to prevent memory defragmentation

FSDP: Industry-grade solution for efficient large model training

# Model Sharding Strategies

## Parameter Shards & Activation Communication

- Parameters remain permanently sharded
- Computations on parameter shards
- Activations communicated between steps


### Limitations

Communication on critical path between dependent computations makes overlapping with computation challenging.

## On-Demand Parameter Communication

- Parameters communicated on-demand
- Each device performs computations as if model fully replicated
- Requires parameters fit in GPU memory during computation


### FSDP's Choice

FSDP adopts this approach because parameter communications don't have data dependencies on preceding computations.

This strategy is sufficient for most current and near-future large model applications

# System Design - Core Architecture

## Decomposition Strategy

- Model instances broken into smaller **FSDP units**
- Each unit contains parameters and gradients
- **Optimizer states** maintained in sharded form

## Memory Management

- Memory proportional to sharded model + largest unit
- Strategic materialization reduces peak memory
- Unsharded parameters discarded after use

## FSDP Architecture Visualization

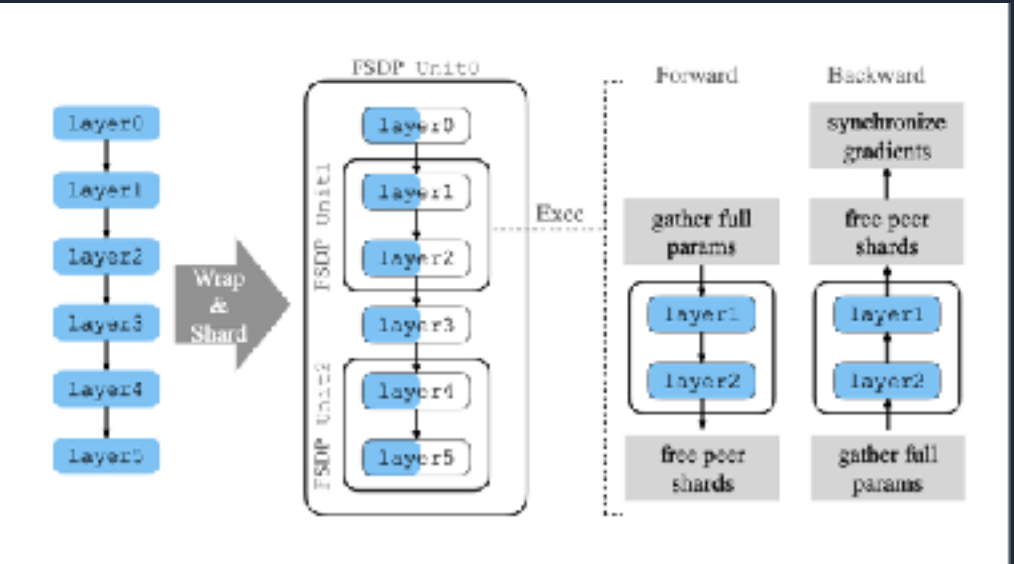



**Figure 1: FSDP Algorithm Overview**

### Key Advantage

FSDP's decomposition approach enables efficient training of models larger than single-device memory by controlling parameter visibility during computation.

# Model Initialization and Deferred Initialization

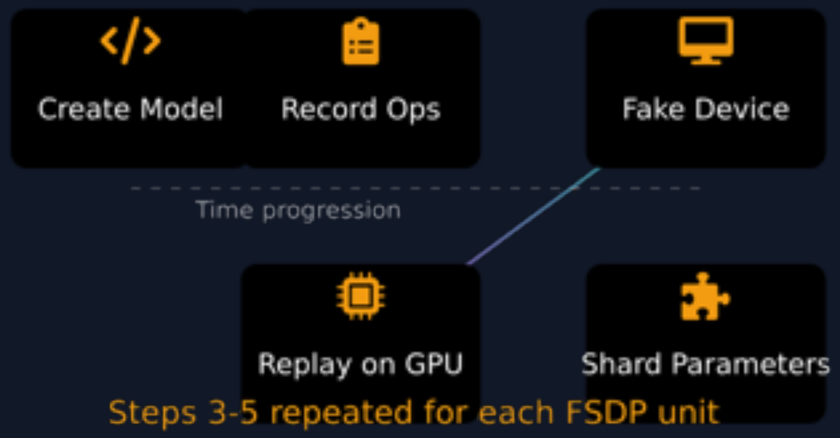
## Initialization Challenge

- Models too large for single GPU memory
- Traditional methods require full model on one device
- Out-of-memory errors with large models

## Deferred Initialization Solution

- Allocate tensors on "fake" device
- Record operations instead of executing
- Replay on real GPU when needed
- Initialize one FSDP unit at a time

## Deferred Initialization Process




### Key Benefits

- Larger models possible
- Memory efficiency
- Cross-submodule dependencies
- Third-party library compatibility

# Sharding Strategies and FlatParameter Design

## Sharding Strategies

### Full Replication (F=1)

F = 1

Model fully replicated across all devices, similar to vanilla data parallelism using AllReduce for gradient reduction.

### Full Sharding (F=W)

F = W

Model completely sharded, with each device holding only 1/W of the model. Minimizes memory footprint but increases communication overhead.

### Hybrid Sharding (1<F<W)

1 < F < W

Combines sharding and replication. Parameters sharded within groups, replicated in complementary groups. Balances memory savings and throughput.

## FlatParameter Design

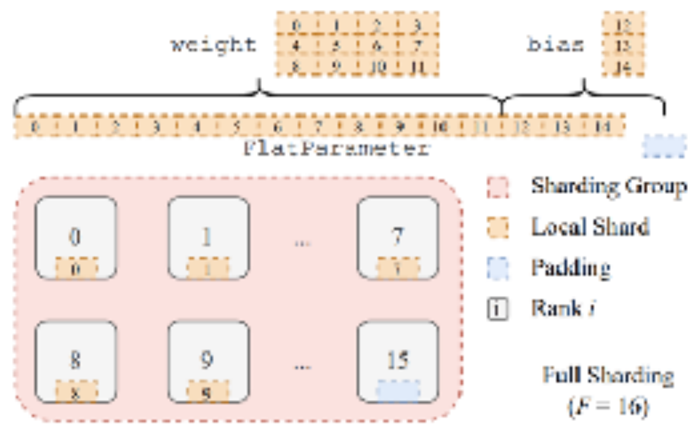



Figure 3: Full Sharding Across 16 GPUs

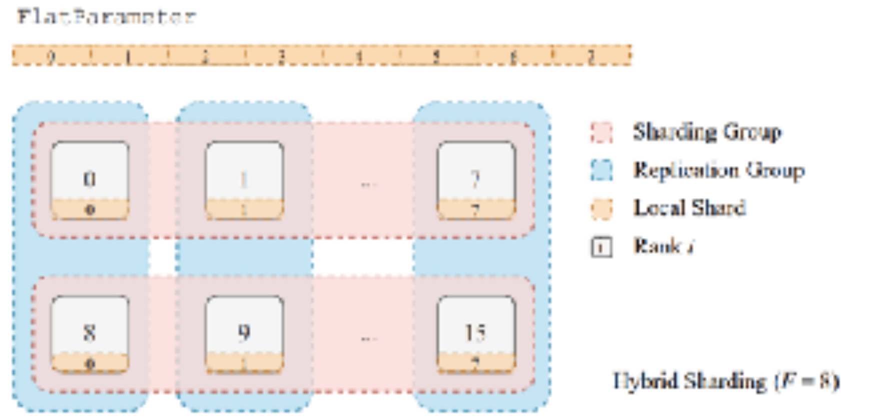



Figure 4: Hybrid Sharding on 16 GPUs: GPUs are configured into 2 sharding groups and 8 replication groups

# Communication Optimizations

### Overlapping Communication & Computation

Uses separate CUDA streams for AllGathers, bypassing false dependencies to allow communication to overlap with computations.

### Backward Prefetching

Issues the next AllGather before current ReduceScatter in backward pass, recording reverse forward order as a proxy for backward order.

### Forward Prefetching

For workloads with static computational graphs, prefetches the next AllGather to help fill potential idle times.

### Gradient Accumulation Variants

Offers two variations: with communication (reducing gradients across ranks) and without (saving unsharded gradients).

### Communication-Computation Overlap

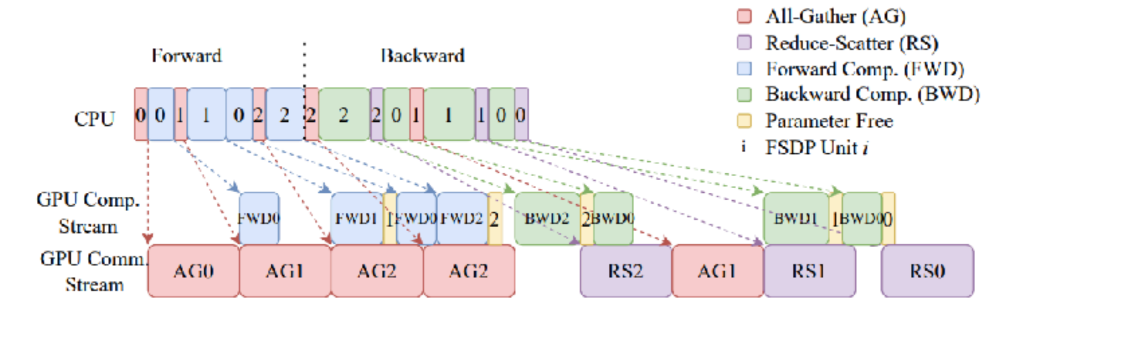



**Figure 5: Overlap Communication and Computation**

# Memory Management and Rate Limiting

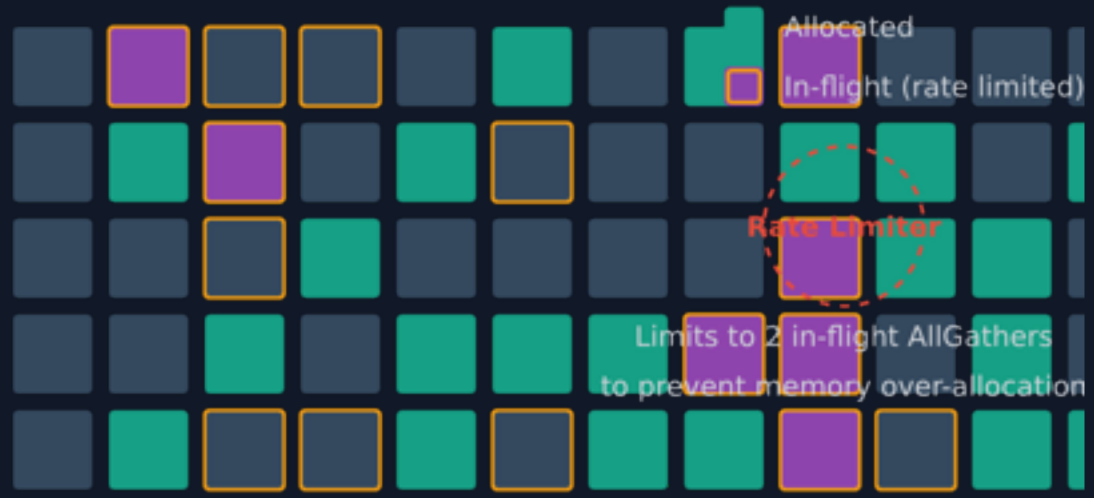
## ⚠ PyTorch's Caching Allocator Challenges

- Frequent memory defragmentations near GPU memory capacity
- Performance degradation with multiple CUDA streams
- Fast CPU threads can run ahead of GPU execution

## FSDP's Rate Limiter Solution

- Intentionally blocks CPU threads to ensure proper caching allocator block reuse
- Limits inflight AllGathers to at most two, minimum required for overlap
- Prevents unnecessary memory over-allocation
- Reduces costly `cudaMalloc` retries

## Memory Fragmentation Visualization




### Key Benefits

- Improved memory utilization
- Reduced memory fragmentation
- Consistent performance scaling
- Better resource allocation

# Implementation Details

## Initialization Options

- Deferred: Allocates model tensors on simulated device, replaying operations on real GPU
- GPU Init: Initialize unsharded model on GPU, then shard with optimizer
- CPU Init: Initialize on CPU with unit-by-unit streaming to GPU

## FlatParameter Design

- Inherits from nn.Parameter with similar behavior
- Managed by FlatParamHandle for FSDP APIs
- Consolidates storage for all parameter tensors within an FSDP unit
- Boundaries dictate timing of AllGather and ReduceScatter

## Runtime Integration with PyTorch

- Forward Hooks: register_forward_pre_hook and register_forward_hook
- Backward Hooks: register_hook on tensor outputs
- Autograd Integration: queue_callback for communication with optimizer

## Native Mixed Precision

- Maintains low-precision (FP16/BF16) and full-precision (FP32) copies
- Uses local sharded FlatParameter to reduce memory overhead
- Independent precision for parameters, gradients, and buffers

FP16/BF16 → Dynamic Alloc → FP32

# Evaluation Results

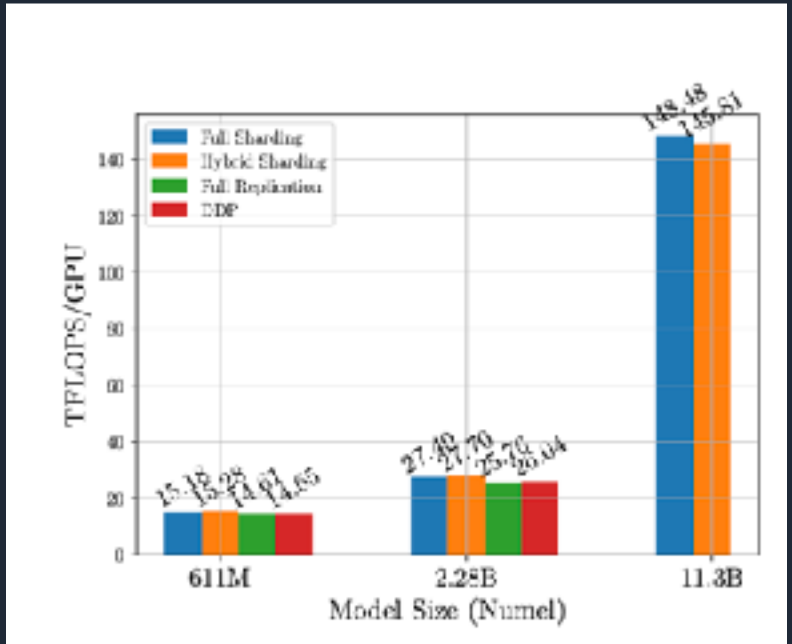
## Scalability Comparison

- **DDP:** Fails for models exceeding **2.28B parameters**
- **Performance:** Backward pre-fetching boosts by **~18%**

## Experimental Setup

- **Hardware:** Up to 512 A100 80GB GPUs
- **Network:** 2Tb/s RoCE network
- **Models:** T5-11B, minGPT-175B

## TFLOPS Performance




(a) Model Scale

### Key Insights

- FSDP achieves **~55-60%** of A100's peak BF16 tensor core utilization
- Near-linear scalability from 128 to 512 GPUs in terms of TFLOPS

# Large Model Scalability Results

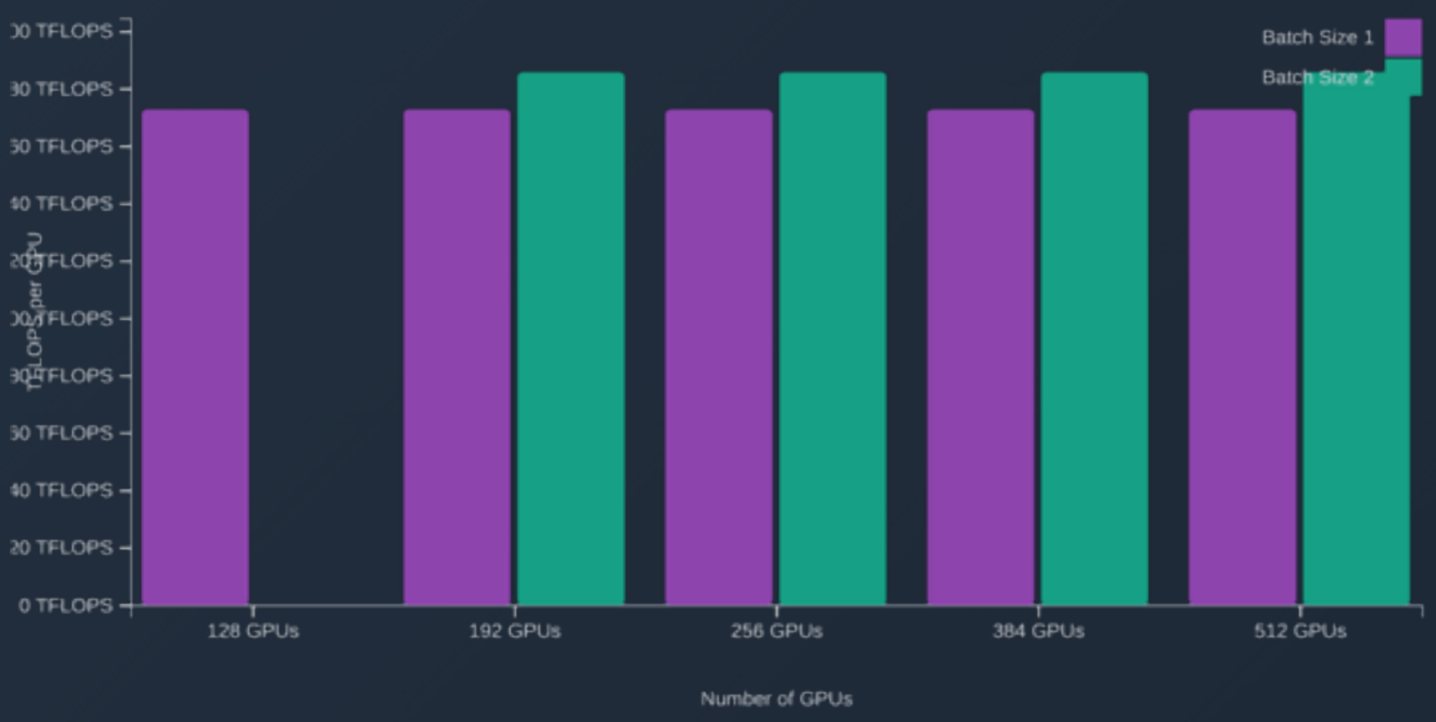
## Key Performance Metrics

- **173+ TFLOPS** per GPU with batch size 1 (~55% utilization)
- **186+ TFLOPS** per GPU with batch size 2 (~60% utilization)
- Near-linear scalability from 128 to 512 GPUs

## Scaling Challenges

- Memory defragmentation with 128 GPUs at batch size 2
- Communication overhead becomes significant with very large clusters

## 175B Model Performance Across GPU Configurations




Performance metrics for minGPT-175B model on A100 80GB GPUs with BF16 precision

# Interoperability and Integration

## Pipeline Parallelism Integration

FSDP can be functionally integrated with pipeline parallelism by wrapping each pipeline stage.

- Default full sharding strategy may incur significant communication overhead due to unsharding for every micro-batch
- Alternative sharding strategies keep parameters unsharded after forward pass
- Reduces unnecessary AllGather communications per micro-batch

⇄ Trades higher memory usage (storing entire pipeline stage parameters) for reduced communication

## Tensor Parallelism Integration

FSDP works well with tensor parallelism to create 2D parallelism for extremely large models.

- Unlike FSDP, tensor parallelism keeps parameters sharded during computation
- Crucial for sub-modules too large to fit in GPU memory
- PyTorch provides

# Limitations and Considerations

## Mathematical Equivalence Challenges

FSDP cannot always guarantee the same mathematical equivalence as local training, especially with optimizer computations.

### Key Issues:

- Optimizer operates on sharded parameters with **FlatParameter** sharding
- Does not respect individual parameter boundaries
- Computations relying on unsharded values become invalid

### Current Approach:

Addressing this often involves uneven sharding or additional communication, which can impact performance. Co-designing optimizer computations with sharding remains an open research question.

## Shared Parameter Handling

FSDP must ensure shared parameters are handled correctly across all usages.

### Key Challenges:

- Shared parameters must not be flattened into multiple **FlatParameters**
- Incorrect handling can lead to errors like missing tensor storage
- Issues if an FSDP unit uses a shared parameter already reshared

### Recommended Solution:

Construct FSDP units such that the shared parameter belongs to the lowest-common-ancestor unit. This ensures the shared parameter remains unsharded throughout its usages.

These limitations are active areas of research

# Conclusion and Future Directions

## FSDP Achievements

### High Usability

Deferred initialization enables creation of models that exceed single-device memory capacity, lowering the barrier for entry.

### Efficiency

Communication overlapping and prefetching techniques maximize GPU utilization and minimize idle times.

### Scalability

Near-linear scalability demonstrated across 128-512 GPUs for models up to 175B parameters.

## Future Directions


### Integration

Enhanced compatibility with pipeline and tensor parallelism.

### Optimization

Refined sharding strategies for heterogeneous hardware.

> FSDP enables efficient training of large models with high usability and efficiency

Thank you