Getting Started with Fully Sharded Data Parallel (FSDP2)

Created On: Mar 17, 2022 | Last Updated: Sep 02, 2025 | Last Verified: Nov 05, 2024

Author: Wei Feng, Will Constable, Yifan Mao

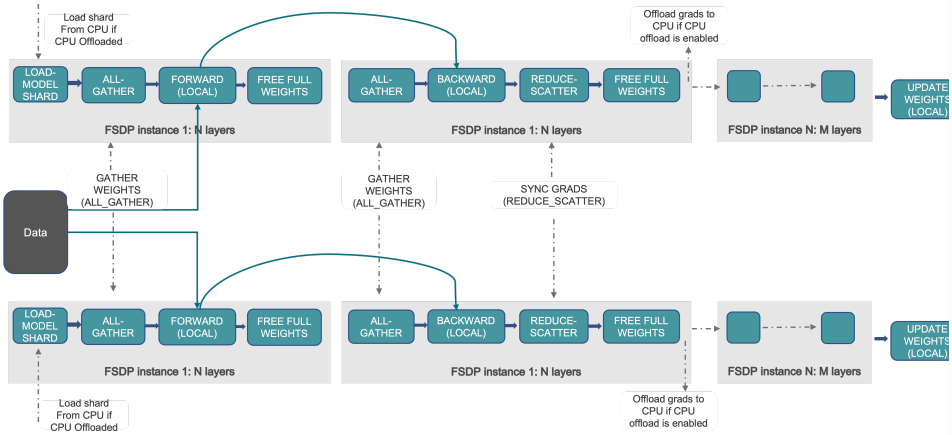
> **Note**
>
> Check out the code in this tutorial from pytorch/examples. FSDP1 is deprecated. FSDP1 tutorials are archived in [1] and [2]

How FSDP2 works

In DistributedDataParallel (DDP) training, each rank owns a model replica and processes a batch of data, finally it uses all-reduce to sync gradients across ranks.

Comparing with DDP, FSDP reduces GPU memory footprint by sharding model parameters, gradients, and optimizer states. It makes it feasible to train models that cannot fit on a single GPU. As shown belo in the picture,

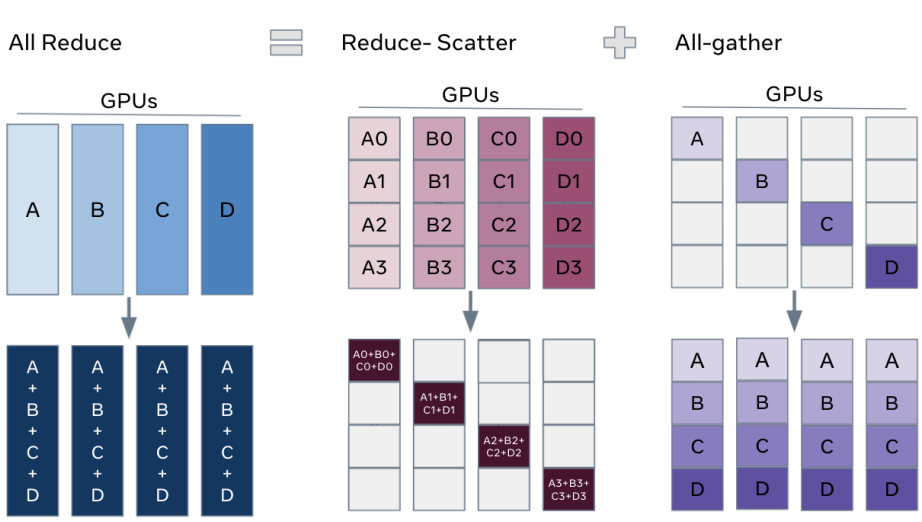
- Outside of forward and backward computation, parameters are fully sharded
- Before forward and backward, sharded parameters are all-gathered into unsharded parameters
- Inside backward, local unsharded gradients are reduce-scatterred into sharded gradients
- Optimizer updates sharded parameters with sharded gradients, resulting in sharded optimizer state



FSDP can be considered a decomposition of DDP's all-reduce into reduce-scatter and all-gather operations

On this page
How FSDP2 works
How to use FSDP2
FSDP1-to-FSDP2 migration guide

PyTorch Libraries
torchao
torchrec
torchft
TorchCodec
torchvision
ExecuTorch
PyTorch on XLA Devices



Comparing with FSDP1, FSDP2 has following advantages:

- Representing sharded parameters as DTensor sharded on dim-i, allowing for easy manipulation of individual parameters, communication-free sharded state dicts, and a simpler meta-device initialization flow.
- Improving memory management system that achieves lower and deterministic GPU memory by avoiding `recordStream` (doc) and does so without any CPU synchronization.
- Offering a tensor subclass extension point to customize the all-gather, e.g. for float8 all-gather for float8 linears (doc), and NF4 for QLoRA (doc)
- Mixing frozen and non-frozen parameters can in the same communication group without using extr memory.

How to use FSDP2

Model Initialization

Applying fully_shard on submodules: Different from DDP, we should apply fully_shard on submodules as well as the root model. In the transformer example below, we applied `fully_shard` on each layer first, then the root model

- During forward computation of `layers[i]`, the rest of the layers are sharded to reduce memory footprint
- Inside `fully_shard(model)`, FSDP2 excludes parameters from `model.layers` and classify remaining parameters into a parameter group for performant all-gather and reduce-scatter
- `fully_shard` moves sharded model to actual training device (eg `cuda`)

Command: `torchrun --nproc_per_node 2 train.py`

```
from torch.distributed.fsdp import fully_shard, FSDPModule
model = Transformer()
for layer in model.layers:
    fully_shard(layer)
fully_shard(model)

assert isinstance(model, Transformer)
assert isinstance(model, FSDPModule)
print(model)
#  FSDPTransformer(
#    (tok_embeddings): Embedding(...)
#    ...
#    (layers): 3 x FSDPTransformerBlock(...)
#    (output): Linear(...)
#  )
```

We can inspect the nested wrapping with `print(model)`. `FSDPTransformer` is a joint class of Transformer and FSDPModule. The same thing happens to FSDPTransformerBlock. All FSDP2 public

APIs are exposed through `FSDPModule`. For example, users can call `model.unshard()` to manually control all-gather schedules. See "explicit prefetching" below for details.

model.parameters() as DTensor: `fully_shard` shards parameters across ranks, and convert `model.parameters()` from plain `torch.Tensor` to DTensor to represent sharded parameters. FSDP2 shards on dim-0 by default so DTensor placements are *Shard(dim=0)*. Say we have N ranks and a parameter with N rows before sharding. After sharding, each rank will have 1 row of the parameter. We can inspect sharded parameters using `param.to_local()`.

```
from torch.distributed.tensor import DTensor
for param in model.parameters():
    assert isinstance(param, DTensor)
    assert param.placements == (Shard(0),)
    # inspect sharded parameters with param.to_local()

optim = torch.optim.Adam(model.parameters(), lr=1e-2)
```

Note the optimizer is constructed after applying `fully_shard`. Both model and optimizer state dicts ar represented in DTensor.

DTensor facilitates optimizer, gradient clipping and checkpointing

- `torch.optim.Adam` and `torch.nn.utils.clip_grad_norm_` works out of the box for DTensor parameters. It makes the code consistent between single-device and distributed training
- we can use DTensor and DCP APIs to manipulate parameters to get full state dict, see "state dict" section below for details. For distributed state dicts, we can save/load checkpoints (doc) without extra communication

Forward/Backward with Prefetching

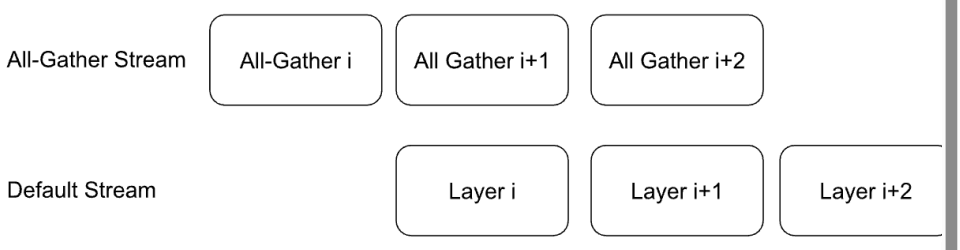
command: `torchrun --nproc_per_node 2 train.py`

```
for _ in range(epochs):
    x = torch.randint(0, vocab_size, (batch_size, seq_len), device=device)
    loss = model(x).sum()
    loss.backward()
    optim.step()
    optim.zero_grad()
```

`fully_shard` registers forward/backward hooks to all-gather parameters before computation, and reshards parameters after computation. To overlap all-gathers with computation, FSDP2 offers **implicit prefetching** that works out of the box with the training loop above and **explicit prefetching** for advanced users to control all-gather schedules manually.

Implicit Prefetching: CPU thread issues all-gather i before layer i. All-gathers are queued into its own cuda stream while layer i computation happens in the default stream. For non-cpu-bound workload (eg Transformer with big batch size), all-gather i+1 can overlap with computation for layer i. Implicit prefetching works similarly in the backward, except all-gathers are issued in the reverse of post-forward order.

> When the CPU thread runs faster, all-gather and computation kernels gets enqueued to cuda streams ahead of time



We recommend users to start with implicit prefetching to understand the performance out of the box.

Explicit Prefetching: Users can specify forward ordering with set_modules_to_forward_prefetch, and backward ordering with set_modules_to_backward_prefetch. As shown in the code below, CPU thread issue all-gather i + 1 and i + 2 at layer i

Explicit prefetching works well in following situation:

CPU-bound workload: If using implicit prefetching, CPU thread will be too slow to issue all-gather for layer i+1 when kernels from layer i get executed. We have to explicitly issue all-gather i+1 before running forward for layer i

Prefetching for 2+ layers: Implicit prefetching only all-gathers next one layer at a time to keep memory footprint minimum. With explicit prefetching can all-gather multiple layers at a time to possibly for better perf with increased memory. See `layers_to_prefetch` in the code

Issuing 1st all-gather earlier: Implicit prefetching happens at the time of calling `model(x)`. The 1st all-gather gets exposed. We can call model.unshard() explicitly earlier to issue 1st all-gather earlier

command: `torchrun --nproc_per_node 2 train.py --explicit-prefetching`

```
num_to_forward_prefetch = 2
for i, layer in enumerate(model.layers):
    if i >= len(model.layers) - num_to_forward_prefetch:
        break
    layers_to_prefetch = [
        model.layers[i + j] for j in range(1, num_to_forward_prefetch + 1)
    ]
    layer.set_modules_to_forward_prefetch(layers_to_prefetch)

num_to_backward_prefetch = 2
for i, layer in enumerate(model.layers):
    if i < num_to_backward_prefetch:
        continue
    layers_to_prefetch = [
        model.layers[i - j] for j in range(1, num_to_backward_prefetch + 1)
    ]
    layer.set_modules_to_backward_prefetch(layers_to_prefetch)

for _ in range(epochs):
    # trigger 1st all-gather earlier
    # this overlaps all-gather with any computation before model(x)
    model.unshard()
    x = torch.randint(0, vocab_size, (batch_size, seq_len), device=device)
    loss = model(x).sum()
    loss.backward()
    optim.step()
    optim.zero_grad()
```

Enabling Mixed Precision

FSDP2 offers a flexible mixed precision policy to speed up training. One typical use case is

- Casting float32 parameters to bfloat16 for forward/backward computation, see `param_dtype=torch.bfloat16`
- Upcasting gradients to float32 for reduce-scatter to preserve accuracy, see `reduce_dtype=torch.float32`

Comparing with torch.amp, FSDP2 mixed precision has following advantages

- **Performant and flexible parameter casting**: All the parameters inside a `FSDPModule` are cast together at the module boundary (before and after before/backward). We can set different mixed precision policies for each layer. For example, the first few layers can be in float32 while remaining layers can be in bfloat16.
- **float32 gradient reduction (reduce-scatter)**: Gradients might vary a lot from rank to rank. Reducing gradients in float32 can be critical for numerics.

command: `torchrun --nproc_per_node 2 train.py --mixed-precision`

```
model = Transformer(model_args)
fsdp_kwargs = {
    "mp_policy": MixedPrecisionPolicy(
        param_dtype=torch.bfloat16,
        reduce_dtype=torch.float32,
    )
}
for layer in model.layers:
    fully_shard(layer, **fsdp_kwargs)
fully_shard(model, **fsdp_kwargs)

# sharded parameters are float32
for param in model.parameters():
    assert param.dtype == torch.float32

# unsharded parameters are bfloat16
model.unshard()
for param in model.parameters(recurse=False):
    assert param.dtype == torch.bfloat16
model.reshard()

# optimizer states are in float32
optim = torch.optim.Adam(model.parameters(), lr=1e-2)

# training loop
# ...
```

Gradient Clipping and Optimizer with DTensor

command: `torchrun --nproc_per_node 2 train.py`

```
# optim is constructed base on DTensor model parameters
optim = torch.optim.Adam(model.parameters(), lr=1e-2)
for _ in range(epochs):
    x = torch.randint(0, vocab_size, (batch_size, seq_len), device=device)
    loss = model(x).sum()
    loss.backward()
    torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=max_norm)
    optim.step()
    optim.zero_grad()
```

Optimizer is initialized after applying `fully_shard` on the model, and holds reference to DTensor `model.parameters()`. For gradient clipping, `torch.nn.utils.clip_grad_norm_` works for DTensor parameters. Tensor ops will be dispatched correctly inside DTensor to communicate partial tensors across ranks to preserve the single device semantic.

State Dicts with DTensor APIs

We showcase how to convert a full state dict into a DTensor state dict for loading, and how to convert it back to full state dict for saving.

command: `torchrun --nproc_per_node 2 train.py`

- For the 1st time, it creates checkpoints for the model and optimizer
- For the 2nd time, it loads from the previous checkpoint to resume training

Loading state dicts: We initialize the model under meta device and call `fully_shard` to convert `model.parameters()` from plain `torch.Tensor` to DTensor. After reading the full state dict from torch.load, we can call distribute_tensor to convert plain `torch.Tensor` into DTensor, using the same placements and device mesh from `model.state_dict()`. Finally we can call model.load_state_dict to load DTensor state dicts into the model.

```
from torch.distributed.tensor import distribute_tensor

# mmap=True reduces CPU memory usage
full_sd = torch.load(
    "checkpoints/model_state_dict.pt",
    mmap=True,
    weights_only=True,
    map_location='cpu',
)
meta_sharded_sd = model.state_dict()
sharded_sd = {}
for param_name, full_tensor in full_sd.items():
    sharded_meta_param = meta_sharded_sd.get(param_name)
    sharded_tensor = distribute_tensor(
        full_tensor,
        sharded_meta_param.device_mesh,
        sharded_meta_param.placements,
    )
    sharded_sd[param_name] = nn.Parameter(sharded_tensor)
# `assign=True` since we cannot call `copy_` on meta tensor
model.load_state_dict(sharded_sd, assign=True)
```

Saving state dicts: `model.state_dict()` returns a DTensor state dict. We can convert a DTensor int
a plain `torch.Tensor` by calling full_tensor(). Internally it issues an all-gather across ranks to get
unsharded parameters in plain torch.Tensor. For rank 0, `full_param.cpu()` offloads the tensor to cpu

PyTorch Distributed Overview

Distributed Data Parallel in PyTorch - Video Tutorials

Getting Started with Distributed Data Parallel

Writing Distributed Applications with PyTorch

Getting Started with Fully Sharded Data Parallel (FSDP2)

Introduction to Libuv TCPStore Backend

Large Scale Transformer model training with Tensor Parallel (TP)

Introduction to Distributed Pipeline Parallelism

Customize Process Group Backends Using Cpp Extensions

Getting Started with Distributed RPC Framework

Implementing a Parameter Server Using Distributed RPC Framework

Implementing Batch RPC Processing Using Asynchronous Executions

Interactive Distributed Applications with Monarch

Combining Distributed DataParallel with Distributed RPC Framework

Distributed Training with Uneven Inputs Using the Join Context Manager


```
sharded_sd = model.state_dict()
cpu_state_dict = {}
for param_name, sharded_param in sharded_sd.items():
    full_param = sharded_param.full_tensor()
    if torch.distributed.get_rank() == 0:
        cpu_state_dict[param_name] = full_param.cpu()
    else:
        del full_param
torch.save(cpu_state_dict, "checkpoints/model_state_dict.pt")
```

Optimizer state dict works similarly (code). Users can customize the above DTensor scripts to work with
3rd party checkpoints.

If there is no need for customization, we can use DCP APIs directly to support both single-node and multi-node training.

State Dict with DCP APIs

command: `torchrun --nproc_per_node 2 train.py --dcp-api`

- For the 1st time, it creates checkpoints for the model and optimizer
- For the 2nd time, it loads from the previous checkpoint to resume training

Loading state dicts: We can load a full state dict into a FSDP2 model with set_model_state_dict. With `broadcast_from_rank0=True`, we can load the full state dict only on rank 0 to avoid peaking CPU memory. DCP will shard tensors and broadcast them to other ranks.

```
from torch.distributed.checkpoint.state_dict import set_model_state_dict
set_model_state_dict(
    model=model,
    model_state_dict=full_sd,
    options=StateDictOptions(
        full_state_dict=True,
        broadcast_from_rank0=True,
    ),
)
```

Saving state dicts: get_model_state_dict with `full_state_dict=True` and `cpu_offload=True` all-gathers tensors and offload them to CPU. It works similarly to DTensor APIs.

```
from torch.distributed.checkpoint.state_dict import get_model_state_dict
model_state_dict = get_model_state_dict(
    model=model,
    options=StateDictOptions(
        full_state_dict=True,
        cpu_offload=True,
    )
)
torch.save(model_state_dict, "model_state_dict.pt")
```

Refer to pytorch/examples for loading and saving optimizer state dicts with set_optimizer_state_dict an get_optimizer_state_dict.

FSDP1-to-FSDP2 migration guide

Let's look at an example of an FSDP usage and an equivalent fully_shard usage. We'll highlight the ke differences and suggest steps for migration.

Original FSDP() usage

```
from torch.distributed.fsdp import FullyShardedDataParallel as FSDP
with torch.device("meta"):
    model = Transformer()
policy = ModuleWrapPolicy({TransformerBlock})
model = FSDP(model, auto_wrap_policy=policy)
def param_init_fn(module: nn.Module) -> None: ...
model = FSDP(model, auto_wrap_policy=policy, param_init_fn=param_init_fn)
```

New fully_shard() usage

```
with torch.device("meta"):
    model = Transformer()
for module in model.modules():
    if isinstance(module, TransformerBlock):
        fully_shard(module)
fully_shard(model)
for tensor in itertools.chain(model.parameters(), model.buffers()):
    assert tensor.device == torch.device("meta")


# Initialize the model after sharding
model.to_empty(device="cuda")
model.reset_parameters()
```

Migration Steps

- Replace the imports
- Implement your 'policy' directly (apply `fully_shard` to the desired sublayers)
- Wrap your root model with `fully_shard` instead of `FSDP`
- Get rid of `param_init_fn` and manually call `model.reset_parameters()`
- Replace other FSDP1 kwargs (see below)

sharding_strategy

- FULL_SHARD: `reshard_after_forward=True`
- SHARD_GRAD_OP: `reshard_after_forward=False`
- HYBRID_SHARD: `reshard_after_forward=True` with a 2D device mesh
- _HYBRID_SHARD_ZERO2: `reshard_after_forward=False` with a 2D device mesh

cpu_offload

- CPUOffload.offload_params=False: `offload_policy=None`
- CPUOffload.offload_params = True: `offload_policy=CPUOffloadPolicy()`

backward_prefetch

- BACKWARD_PRE: always used

- BACKWARD_POST: not supported

mixed_precision