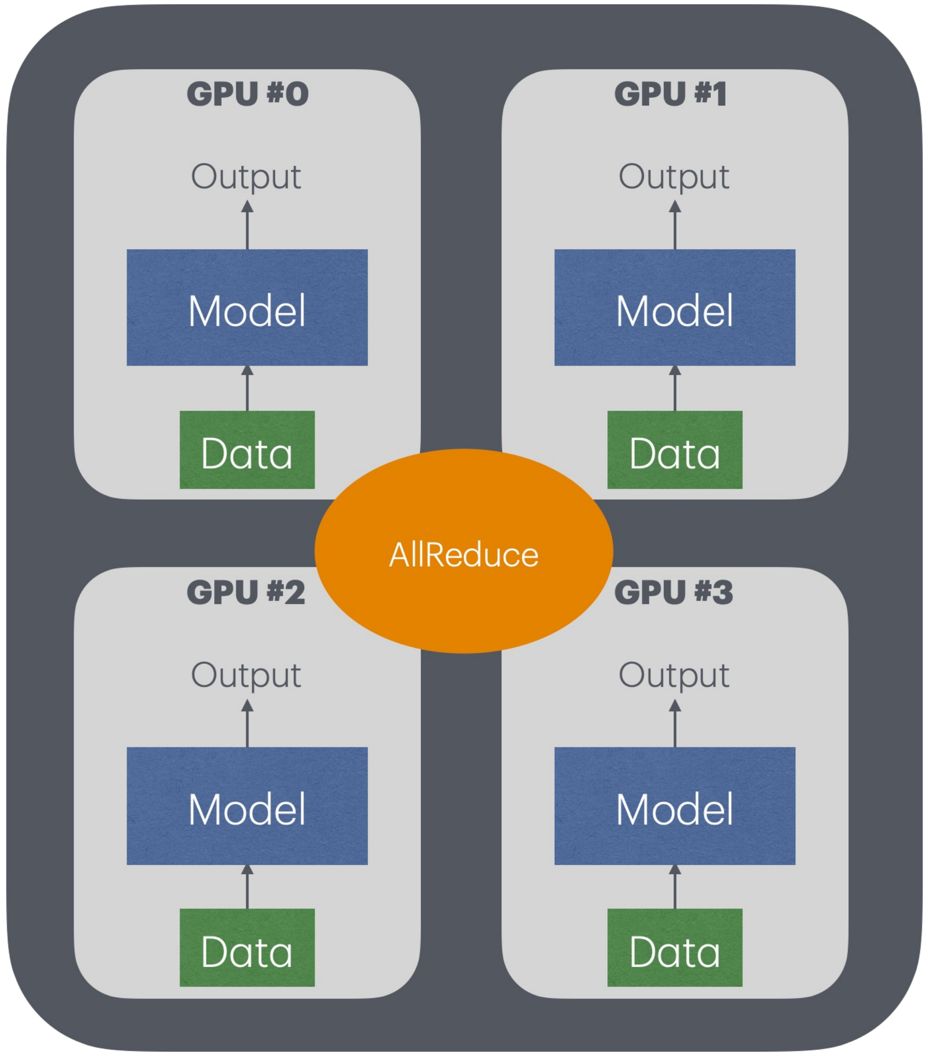
TRAINING DEEP LEARNING MODELS WITH FULLY SHARDED DATA PARALLEL

SABRINA BENASSOU

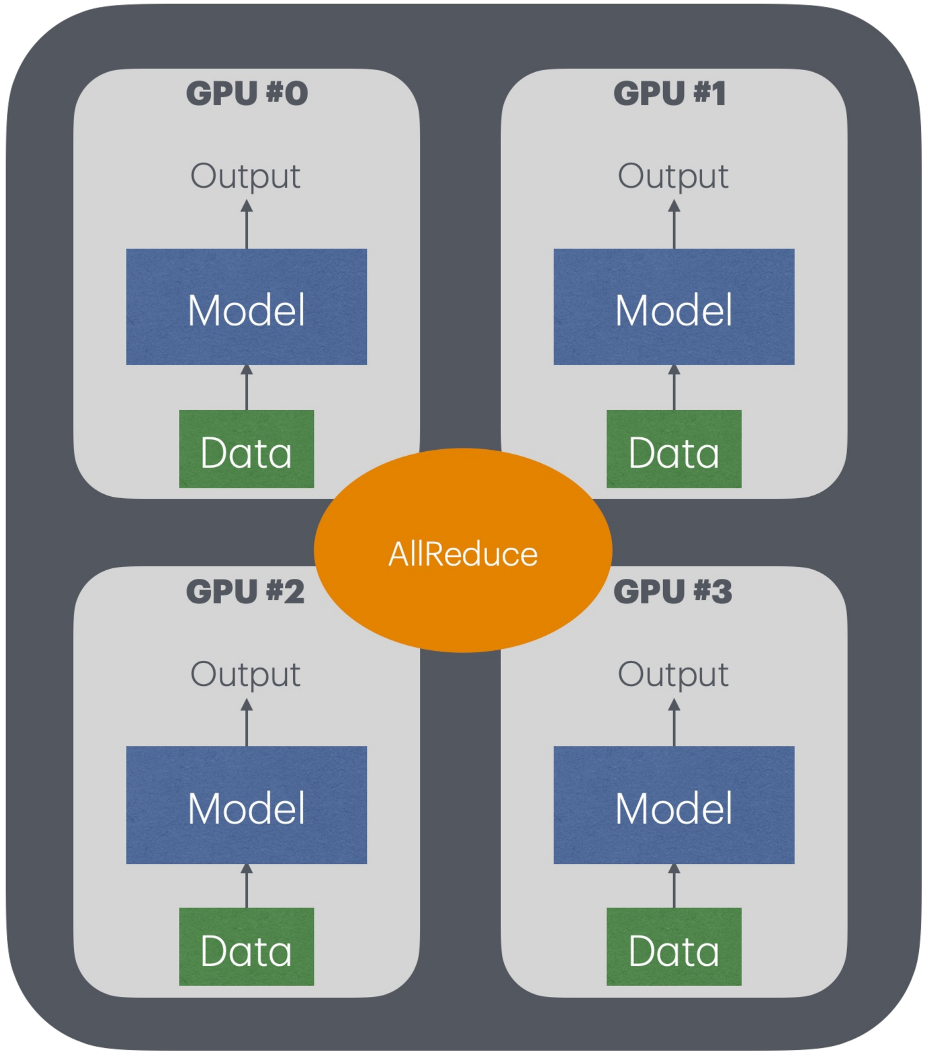
SUMMARY

SUMMARY

DISTRIBUTED DATA PARALLEL (DDP)

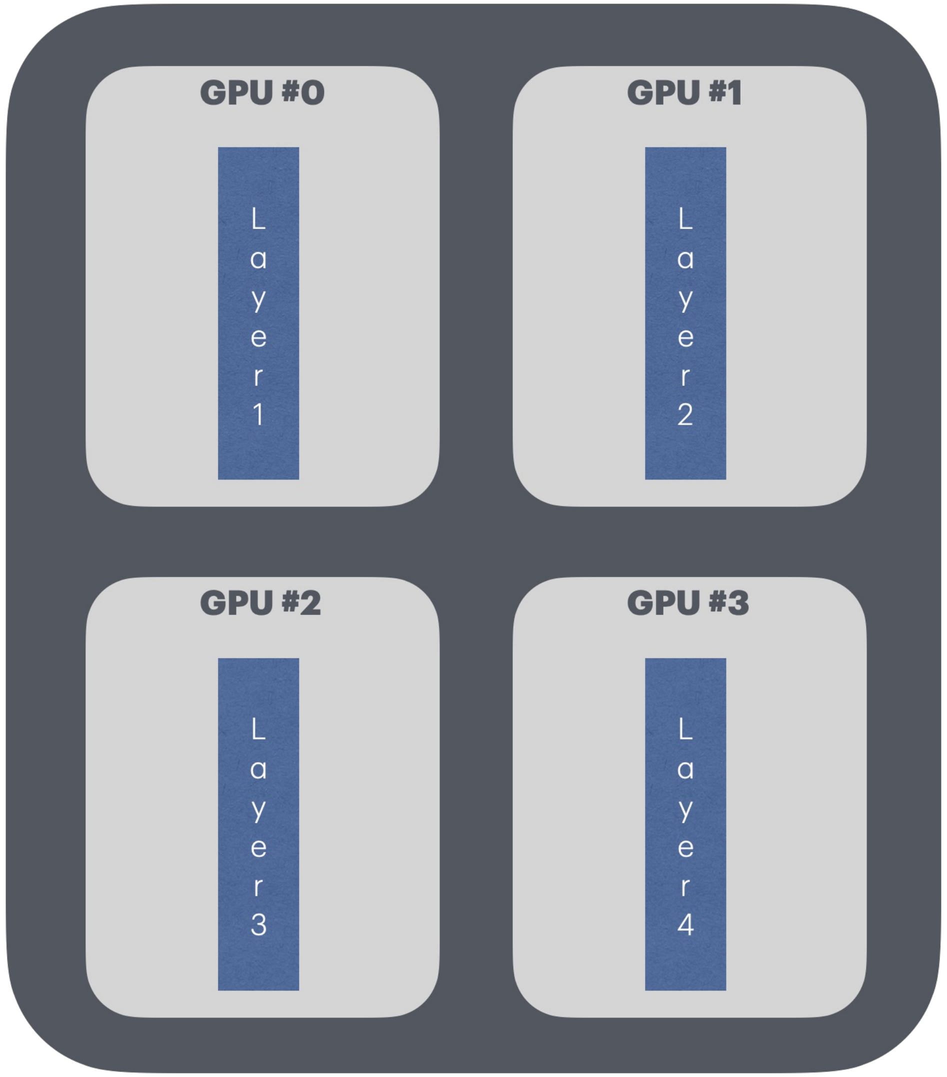


DISTRIBUTED DATA PARALLEL (DDP)

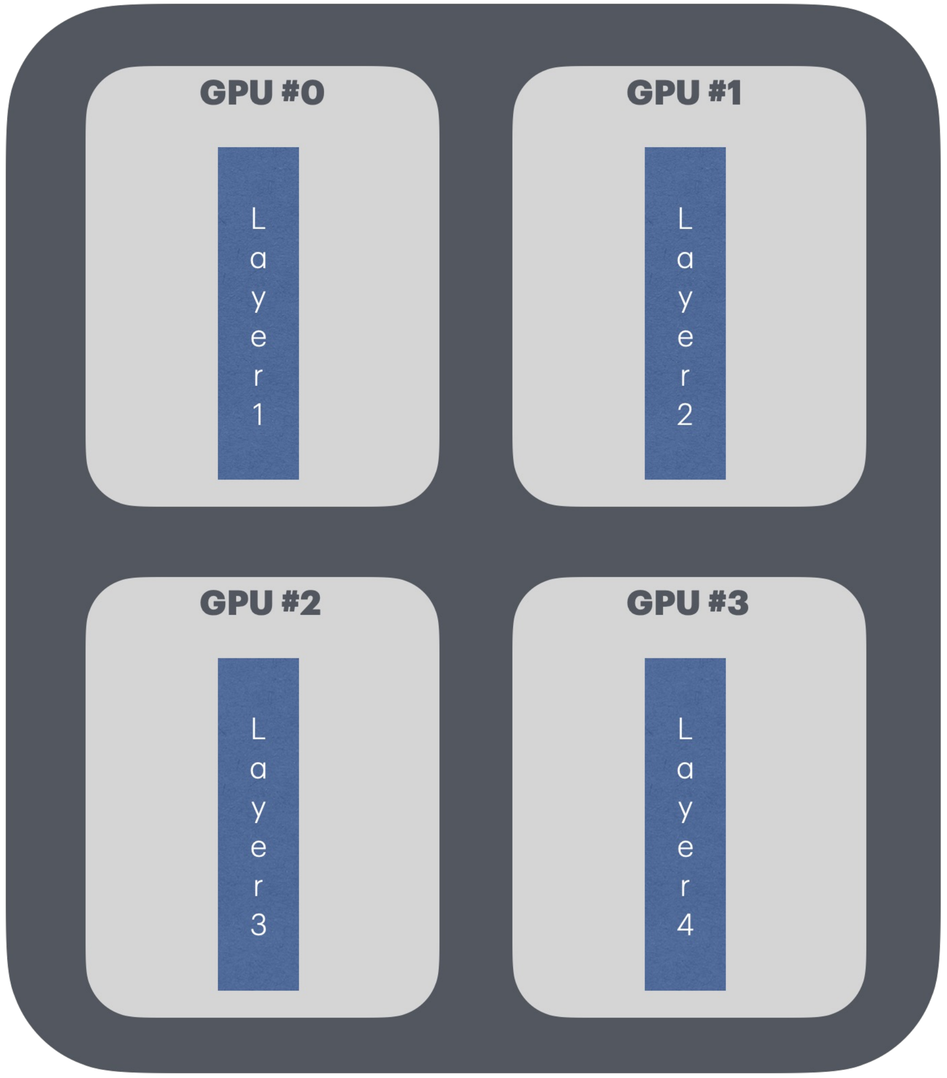


- Inadequate for supporting large models
- Likely encounter out-of-memory errors on each device

PIPELINE PARALLELISM



PIPELINE PARALLELISM



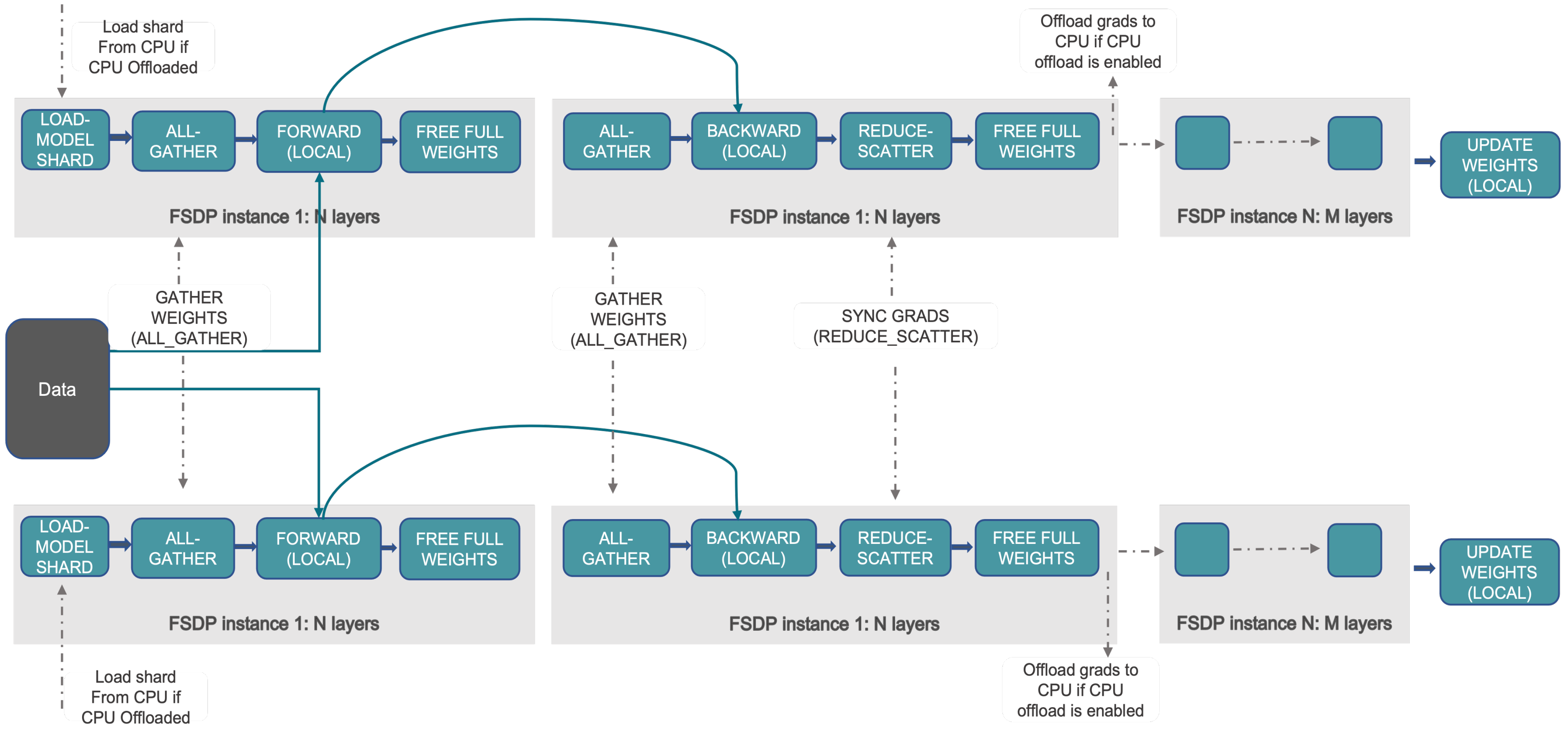
- The entire model must be structured in sequential layers or stages, which might not be feasible for all types of models.

OTHER PARALLELISM STRATEGIES

- Tensor Parallelism
- ZeRo
- MiCS
- …

FULLY SHARDED DATA PARALLEL (FSDP)

- Distributed training technique used to scale deep learning model training across multiple GPUs by sharding the model's parameters.
- Integrates efficiently with PyTorch and optimizes communication and memory usage, making it effective for training extremely large models in a distributed environment.


Load shard From CPU if CPU Offloaded
LOAD-MODEL SHARD
ALL-GATHER
FORWARD (LOCAL)
FREE FULL WEIGHTS
FSDP instance 1: N layers
ALL-GATHER
BACKWARD (LOCAL)
REDUCE-SCATTER
FREE FULL WEIGHTS
FSDP instance 1: N layers
Offload grads to CPU if CPU offload is enabled
FSDP instance N: M layers
UPDATE WEIGHTS (LOCAL)
GATHER WEIGHTS (ALL_GATHER)
Data
GATHER WEIGHTS (ALL_GATHER)
SYNC GRADS (REDUCE_SCATTER)
LOAD-MODEL SHARD
ALL-GATHER
FORWARD (LOCAL)
FREE FULL WEIGHTS
FSDP instance 1: N layers
ALL-GATHER
BACKWARD (LOCAL)
REDUCE-SCATTER
FREE FULL WEIGHTS
FSDP instance 1: N layers
FSDP instance N: M layers
UPDATE WEIGHTS (LOCAL)
Load shard From CPU if CPU Offloaded
Offload grads to CPU if CPU offload is enabled

ALGORITHM OVERVIEW

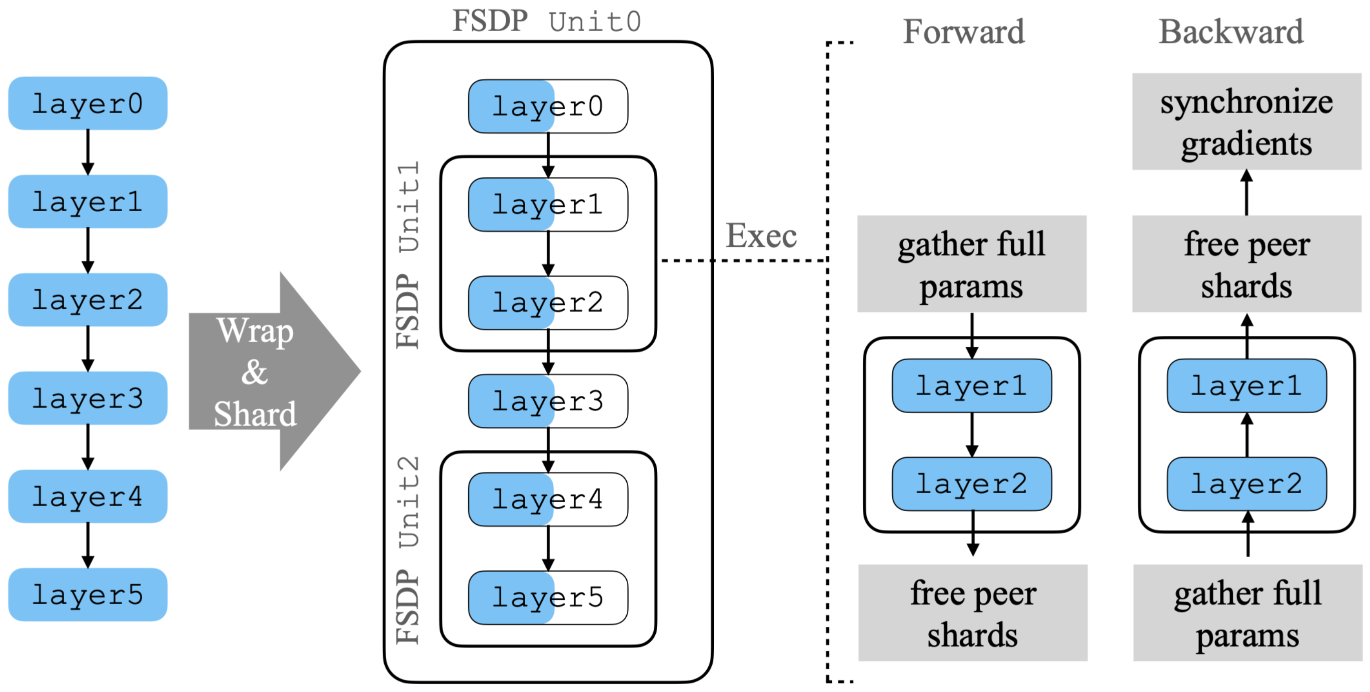


Figure 1: FSDP Algorithm Overview

COMMUNICATION OPTIMIZATIONS

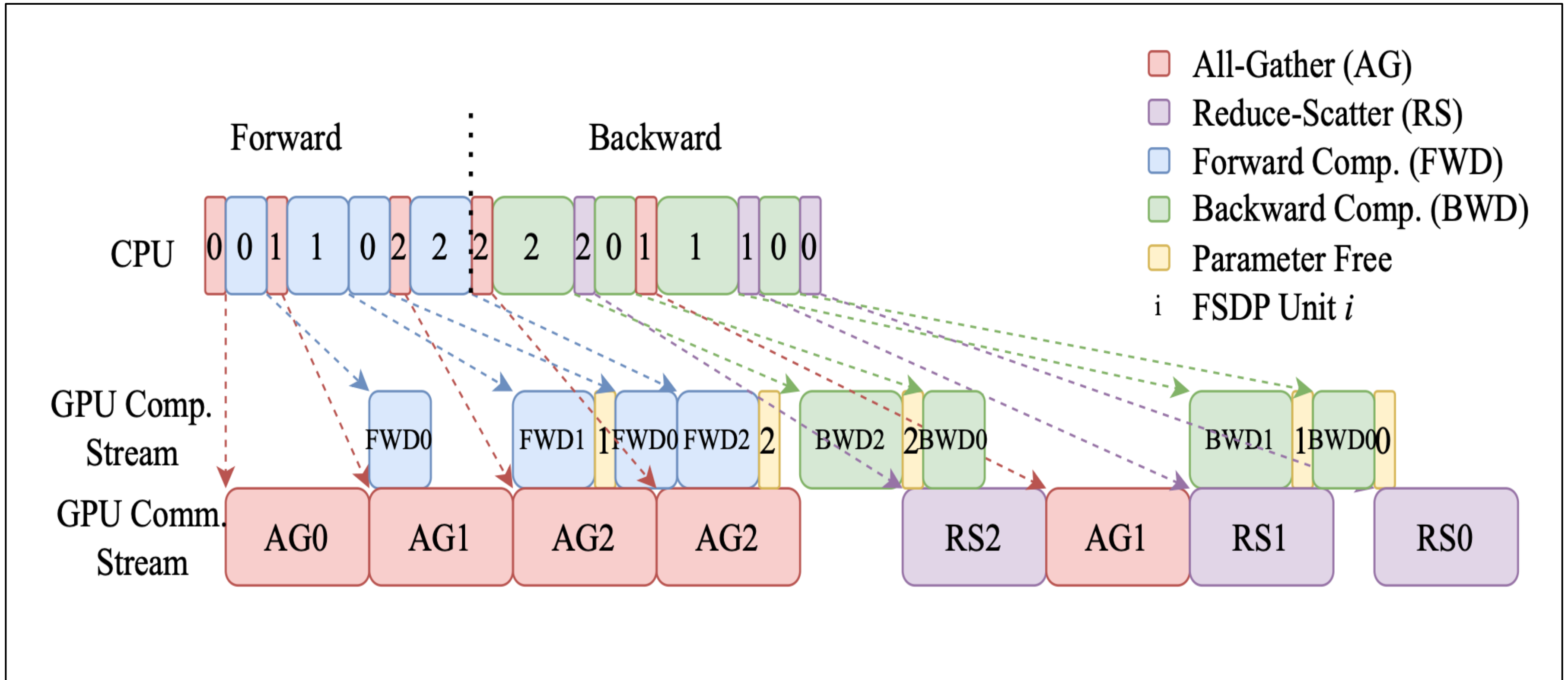
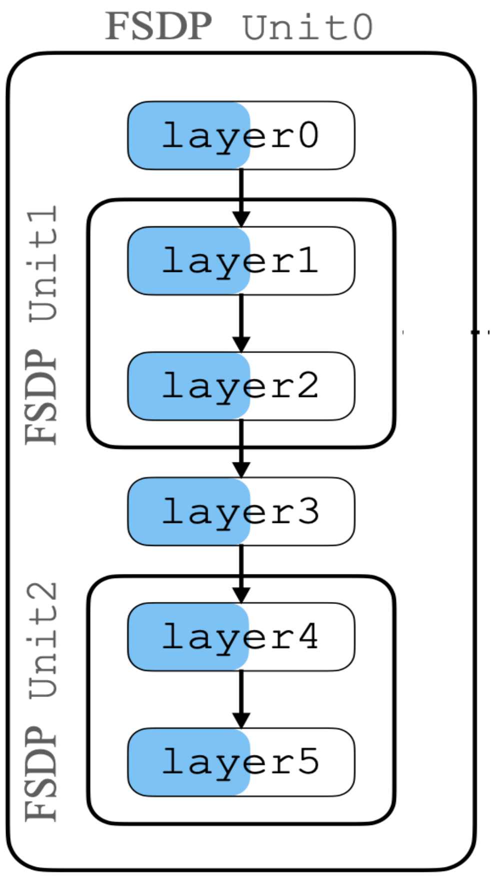


Figure 5: Overlap Communication and Computation



DEFERRED INITIALIZATION

Challenge #1:

- PyTorch required full materialization of the entire model on one device, 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.

Solution #1:

- Model parameters are initially allocated on a simulated or "fake" device.
- All parameter initialization operations are recorded during this simulated phase.
- When parameters move to a GPU, recorded initialization operations are replayed.
- This method creates a model instance without consuming GPU memory initially.

MODEL INITIALIZATION

Challenge #2

- How to ensure accurate initialization of model parameters when the model is too large to fit on a single GPU.

Solution #2:

- FSDP initializes and shards one unit at a time.
- With deferred initialization, FSDP decomposes the model into units, moves each unit to a GPU sequentially, and replays tensor initialization operations.

CODE DEMO

GOOD PRACTICE

- Always store your code in the project folder.

```
mkdir /p/project/training2426/$USER
cd /p/project/training2426/$USER
git clone https://github.com/sab148/nxtaim-distributed-models
```

- Store data in the scratch directory for faster job access. **Files in scratch are deleted after 90 days of inactivity.**

```
ls /p/scratch/training2426/
```

- Store the data in \`$DATA_dataset\` for a more permanent location.
  - This location is not accessible by compute nodes.
  - You have to Join the project in order to store and access data https://judoor.fz-juelich.de/projects/datasets/

LET’S PARALLELIZE THE CODE WITH DISTRIBUTED DATA PARALLEL (DDP)

Remove line 180 and add this code snippet at line 172

```python
# Initializes a communication group using 'nccl' as the backend for GPU communication.
torch.distributed.init_process_group(backend='nccl')

# Get the identifier of each process within a node
local_rank = int(os.getenv('LOCAL_RANK'))

# Get the global identifier of each process within the distributed system
rank = int(os.environ['RANK'])

# Creates a torch.device object that represents the GPU to be used by this process.
device = torch.device('cuda', local_rank)

# Sets the default CUDA device for the current process,
# ensuring all subsequent CUDA operations are performed on the specified GPU device.
torch.cuda.set_device(device)

# Different random seed for each process.
torch.random.manual_seed(args.seed + torch.distributed.get_rank())
```

Add this snippet at line 111

```
# Ensures that each process gets a unique subset of the data.
# shuffle=True only for training set
train_sampler = torch.utils.data.distributed.DistributedSampler(
    train_dset,
    shuffle=True,
    seed=args.seed,
)

train_dset = torch.utils.data.DataLoader(
    train_dset,
    batch_size=args.batch_size,
    # Uses the distributed sampler to ensure each process gets a different subset of the data.
    sampler=train_sampler,
    # Use multiple processes for loading data.
    num_workers=args.train_num_workers,
    # Use pinned memory on GPUs for faster device-copy.
    pin_memory=True,
    persistent_workers=args.train_num_workers > 0,
)
```

Don't forget this line

```
# Ensures that each process gets a unique subset of the data.
valid_sampler = torch.utils.data.distributed.DistributedSampler(valid_dset)

valid_dset = torch.utils.data.DataLoader(
    valid_dset,
    batch_size=args.batch_size,
    # Uses the distributed sampler to ensure each process gets a different subset of the data.
    sampler=valid_sampler,
    num_workers=args.valid_num_workers,
    # Use pinned memory on GPUs for faster device-copy.
    pin_memory=True,
    persistent_workers=args.valid_num_workers > 0,
)
```

Here as well

```
# Ensures that each process gets a unique subset of the data.
test_sampler = torch.utils.data.distributed.DistributedSampler(test_dset)

test_dset = torch.utils.data.DataLoader(
    test_dset,
    batch_size=args.batch_size,
    # Uses the distributed sampler to ensure each process gets a different subset of the data.
    sampler=test_sampler,
    # Use pinned memory on GPUs for faster device-copy.
    pin_memory=True,
)
```

Add this snippet at line 210

```python
# Wraps the model in a DistributedDataParallel (DDP) module to parallelize the training
# across multiple GPUs.
model = torch.nn.parallel.DistributedDataParallel(
    model,
    device_ids=[local_rank],
)
```

Add at line 227

```
# sets the current epoch for the dataset sampler to ensure proper data shuffling in each epoch
train_dset.sampler.set_epoch(epoch)
```

Add this line of code at line 242 and line 176

```
# Obtain the global average loss.
torch.distributed.all_reduce(loss, torch.distributed.ReduceOp.AVG)
```

Add those utility functions at line 85

```python
functools.lru_cache(maxsize=None)
def is_root_process():
    """Return whether this process is the root process."""
    return torch.distributed.get_rank() == 0


def print0(*args, **kwargs):
    """Print something only on the root process."""
    if is_root_process():
        print(*args, **kwargs)


def save0(*args, **kwargs):
    """Pass the given arguments to `torch.save`, but only on the root
    process.
    """
    # We do *not* want to write to the same location with multiple
    # processes at the same time.
    if is_root_process():
        torch.save(*args, **kwargs)
```

Replace print with print0 at line 269

```
print0(f'[{epoch}/{args.epochs}; {i}] loss: {loss:.5f}')
```

Replace print with print0 at line 276

```
print0(f'[{epoch}/{args.epochs}; {i}] valid loss: {valid_loss:.5f}')
```

Replace print with print0 at line 290

```
print0('Finished training after', end_time - start_time, 'seconds.')
```

Replace print with print0 at line 296

```
print0('Final test loss:', test_loss)
```

Replace torch.save at line 283

```python
# Replace torch.save function by the utility function to save the model.
save0(model, 'model-best.pt')
```

Replace torch.save at line 300

```python
# Replace torch.save function by the utility function to save the model.
save0(model, 'model-final.pt')
```

Change number of node at line 3 in run_to_distributed_training.sh file

```
#SBATCH --nodes=4
```

Add in run_to_distributed_training.sh file at line 17

```
# so processes know who to talk to
echo "SLURM_JOB_NODELIST: $SLURM_JOB_NODELIST"
MASTER_ADDR="$(scontrol show hostnames "$SLURM_JOB_NODELIST" | head -n 1)"
# Allow communication over InfiniBand cells.
MASTER_ADDR="${MASTER_ADDR}i"
# Get IP for hostname.
export MASTER_ADDR="$(nslookup "$MASTER_ADDR" | grep -oP '(?<=Address: ).*')"
export MASTER_PORT=7010
export GPUS_PER_NODE=4

echo "MASTER_ADDR:MASTER_PORT=""$MASTER_ADDR":"$MASTER_PORT"
echo "----------------------------------"
```

Add in run_to_distributed_training.sh file at line 41

```
srun --cpu_bind=none bash -c "torchrun \
--nnodes=$SLURM_NNODES \
--rdzv_backend c10d \
--nproc_per_node=gpu \
--rdzv_id $RANDOM \
--rdzv_endpoint=$MASTER_ADDR:$MASTER_PORT \
--rdzv_conf=is_host=\$(if ((SLURM_NODEID)); then echo 0; else echo 1; fi) \
to_distributed_training.py "
```

JÜLICH Forschungszentrum

PARALLELIZE THE CODE WITH FULLY SHARDED DATA PARALLEL (FSDP)

Replace DistributedDataParallel with FullyShardedDataParallel at line 231

```
# `my_auto_wrap_policy` function will automatically wraps submodules with at least
20,000 parameters for distributed training.
my_auto_wrap_policy = functools.partial(
    size_based_auto_wrap_policy, min_num_params=20000
)

# Wraps the model in FullyShardedDataParallel (FSDP) module using the specified GPU and
automatic wrapping policy for submodules.
model = fsdp.FullyShardedDataParallel(
    model,
    device_id=local_rank,
    auto_wrap_policy=my_auto_wrap_policy,
)
```

Add this function at line 85

```python
def save_model(
     checkpoint_type,
     model,
     rank,
     save_dir='model-final.pt',
     optim_dir="optimizer-final.pt",
     optimizer=None
):

     if checkpoint_type == 'full':
          model_checkpointing.save_model_checkpoint(model, save_dir, rank)
     if optimizer is not None:
          model_checkpointing.save_optimizer_checkpoint(model, optimizer, optim_dir, rank)

     elif checkpoint_type == 'sharded':
          if optimizer is not None:
               model_checkpointing.save_model_and_optimizer_sharded(model, save_dir, rank,
     optim=optimizer)
          else:
               model_checkpointing.save_model_and_optimizer_sharded(model, save_dir, rank)

     elif checkpoint_type == 'local':
          model_checkpointing.save_distributed_model_checkpoint(model, save_dir, rank)
          if optimizer is not None:
               model_checkpointing.save_optimizer_checkpoint(model, optimizer, optim_dir, rank)
```

Replace save0 at line 312

```
# Replace the following line with the utility function to save the model.
save_model(args.save_model_opt, model, rank, "model-best.pt", "optimizer-best.pt",
optimizer=opt)
```

Replace save0 at line 329

```
# Replace the following line with the utility function to save the model.
save_model(args.save_model_opt, model, rank, optimizer=opt)
```

Export TORCH_LOGS at line 37 in run_to_distributed_training.sh file

```
export TORCH_LOGS='-torch.distributed.checkpoint._dedup_tensors'
```

SHARDING STRATEGIES

SHARDING STRATEGIES

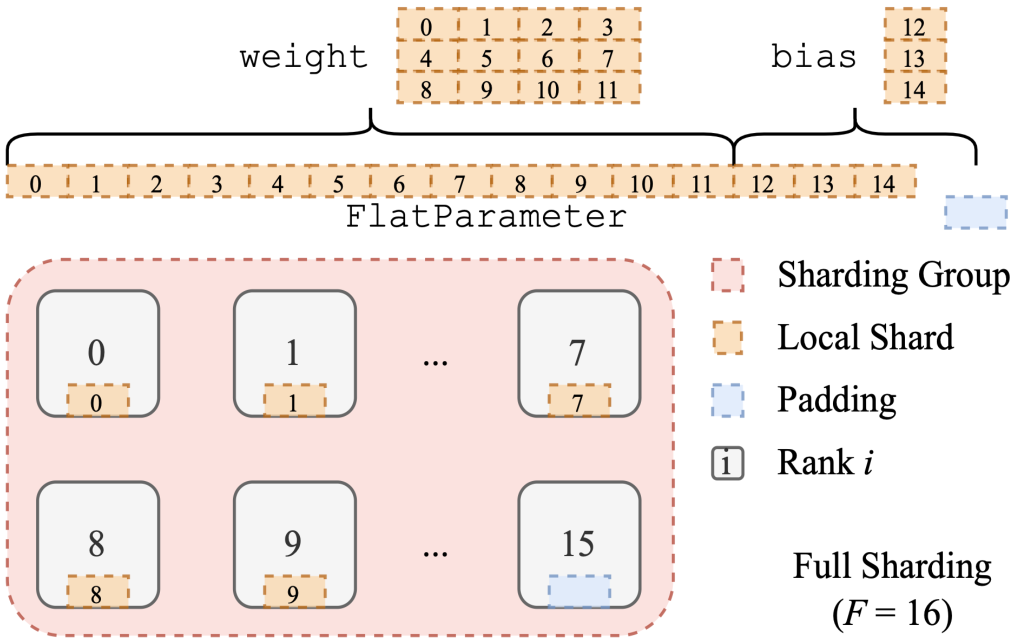


Figure 3: Full Sharding Across 16 GPUs

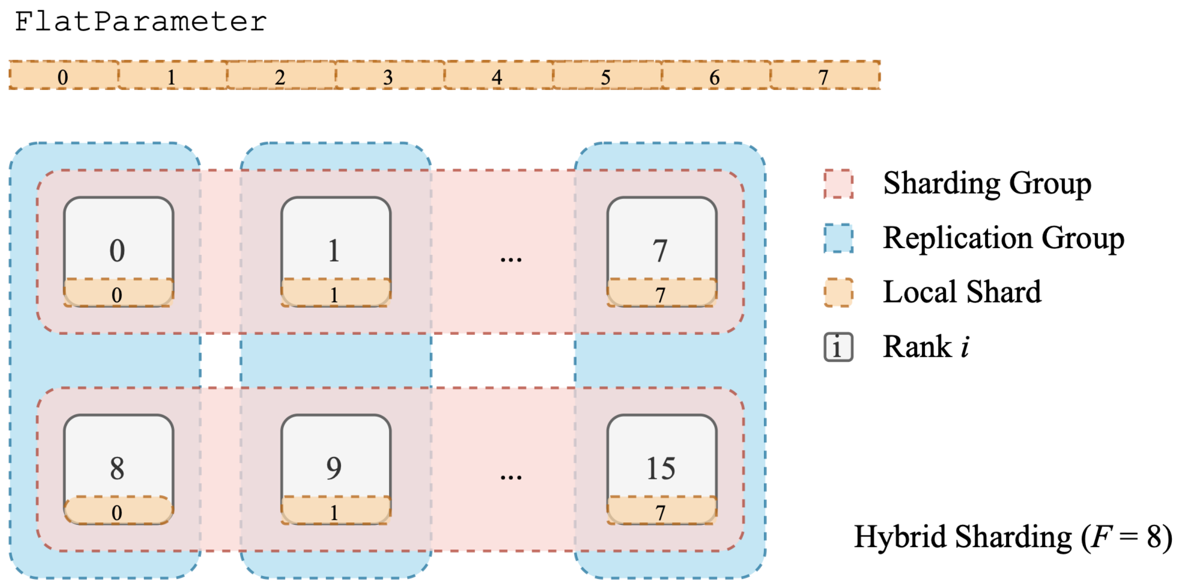


Figure 4: Hybrid Sharding on 16 GPUs: GPUs are configured into 2 sharding groups and 8 replication groups

SHARDING STRATEGIES

Sharding strategies for distributed training by FullyShardedDataParallel

- `FULL_SHARD`: Parameters, gradients, and optimizer states are sharded. Default strategy.
- `SHARD_GRAD_OP`: For the parameters, this strategy unshards before the forward, does not reshard them after the forward, and only reshards them after the backward computation
- `NO_SHARD`: Similar to PyTorch's `DistributedDataParallel` API.
- `HYBRID_SHARD`: Apply `FULL_SHARD` within a node, and replicate parameters across nodes.
- `_HYBRID_SHARD_ZERO2`: Apply `SHARD_GRAD_OP` within a node, and replicate parameters across nodes. This is like `HYBRID_SHARD`, except this may provide even higher throughput since the unsharded parameters are not freed after the forward pass, saving the all-gathers in the pre-backward.

Pass this argument at line 260

```
# Wraps the model in FullyShardedDataParallel (FSDP) module using the specified GPU and automatic wrapping policy for submodules.
model = fsdp.FullyShardedDataParallel(
    model,
    device_id=local_rank,
    auto_wrap_policy=my_auto_wrap_policy,
    sharding_strategy=fsdp.ShardingStrategy.HYBRID_SHARD
)
```

FSDP INTEROPERABILITY

Combination with Pipeline Parallelism

- **Integration**: FSDP can wrap each stage of a pipeline parallel model.
- **Challenge with Micro-Batches**: Pipeline parallelism involves splitting input batches into micro-batches, requiring frequent unsharding and resharing of model parameters, which can lead to high communication overhead.
- **Optimization**: FSDP offers alternative sharding strategies that keep parameters unsharded post-forward pass, reducing the need for AllGather communications after each micro-batch.
- **Memory Efficiency**: Even though entire pipeline stage parameters are stored on the GPU device, FSDP minimizes memory usage by still sharding gradients and optimizer states.

FSDP INTEROPERABILITY

Combination with Tensor Parallelism

- **Distinctive Approach**: Unlike FSDP, tensor parallelism maintains parameters sharded during computations to fit large sub-modules within GPU memory constraints.
- **2D Parallelism in PyTorch**: PyTorch's parallelize_module feature can create a 2D parallel structure by combining tensor parallelism and FSDP.
- **Device Organization**: Devices are organized into a 2D mesh, with DTensor managing tensor parallelism on one dimension (typically intra-node for higher bandwidth) and FSDP handling sharded data parallelism on the other dimension (inter-node).
- **Communication Strategy**: This setup optimizes communication paths, keeping tensor-parallel operations within the node and FSDP communications between nodes to efficiently manage network bandwidth and computational delays.

LIMITATIONS

Mathematical Equivalence:

- **Challenges with Optimizer Computation**: FSDP's sharding mechanism can alter the original data layout of parameters, impacting computations that depend on unsharded values, such as vector norms or complex optimizer functions.
- **Impact on Optimizer Efficiency**: Using padding, uneven sharding, or extra communication to maintain optimizer accuracy can negatively affect performance.
- **Research Area**: Properly integrating sophisticated optimizer computations with parameter sharding remains an open and active area of research.

Shared Parameters:

- **Management of Shared Parameters**: FSDP must avoid flattening shared parameters into multiple segments to prevent errors related to tensor storage or size mismatches.
- **Strategic Sharding**: It's recommended to structure FSDP units so that shared parameters are always accessible and unsharded across all uses by placing them in the lowest-common-ancestor unit.
- **Ongoing Improvements**: Methods to handle shared parameters more efficiently are under investigation to optimize their management without extended periods of being unsharded.

CONCLUSION

- FSDP is a distributed training method in PyTorch that shards model parameters and optimizer states across GPUs to maximize memory efficiency and scalability.
- FSDP enables model initialization on a dummy device, which is then replayed on GPUs, supporting large models. This is called deferred initialization.
- FSDP decomposes the model into units and materializes the parameters of one unit at a time.
- FSDP offers different sharding strategies.
- FSDP can be combined with pipeline parallelism or tensor parallelism.

REFERENCE

- Zhao, Yanli, et al. "Pytorch fsdp: experiences on scaling fully sharded data parallel." *arXiv preprint arXiv:2304.11277* (2023).

Thank You!