

FSDP Lecture

Contents

- Motivation for developing another sharding method
- FSDP basic implementation
- Why is FSDP almost free?
 - NCCL Crash Course
 - Bytes transferred in FSDP vs DDP
 - Torch Profiler crash course
 - Forward and backward prefetching
- Flavors of FSDP
 - FULL_SHARD
 - SHARD_GRAD_OP
 - HYBRID_SHARD
- FSDP implementations and “how bro are they?”

Modern DL model sizes

- Most models are bigger than 7B
- LLaMA family: 7B, 70B, 405B

Trending last 7 days

All Models Datasets Spaces

deepseek-ai/DeepSeek-R1

Text Generation • Update... • 3.21M • 8.61k

Zyphra/Zonos-v0.1-hybrid

Text-to-Speech • Updated about... • 2.76k • 675

open-r1/OpenR1-Math-220k

Viewer • Updated abou... • 450k • 1.93k • 223

hexgrad/Kokoro-82M

Text-to-Speech • Updated 11 day... • 479k • 3.1k

deepseek-ai/Janus-Pro-7B

Any-to-Any • Updated 12 days ago • 391k • 2.92k

open-thoughts/OpenThoughts-114k

Viewer • Updated abou... • 228k • 51.4k • 447

simplescaling/s1K

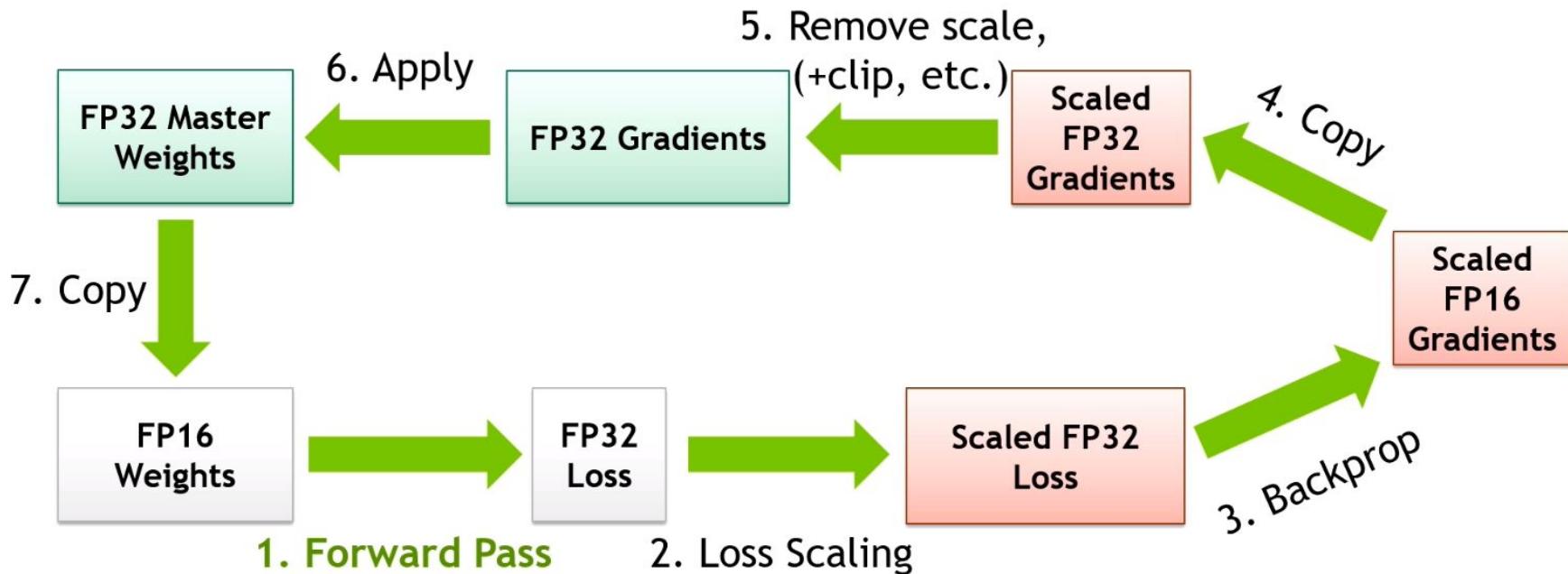
Viewer • Updated 2 days ... • 1k • 2.98k • 162

fka/awesome-chatgpt-prompts

Viewer • Updated Jan 6 • 203 • 11k • 7.46k

Mixed Precision recap

MIXED PRECISION TRAINING



Mixed Precision recap

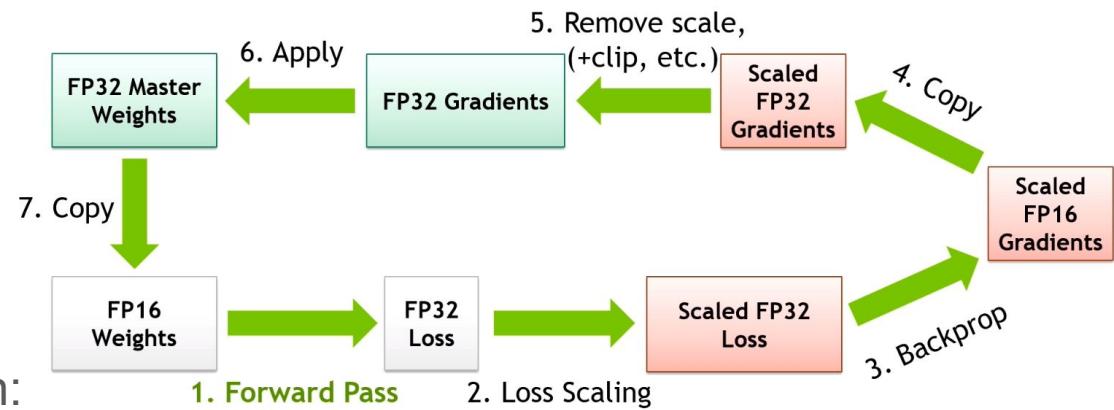
Static memory consumption:

- FP32 master weights and Adam moments (4 bytes x 3 buffers)
- FP16 copy of weights for calculations (2 bytes)
- FP16 gradients (2 bytes)

Dynamic memory consumption:

- Activations (depends on batch size, etc)

MIXED PRECISION TRAINING



Mixed Precision recap

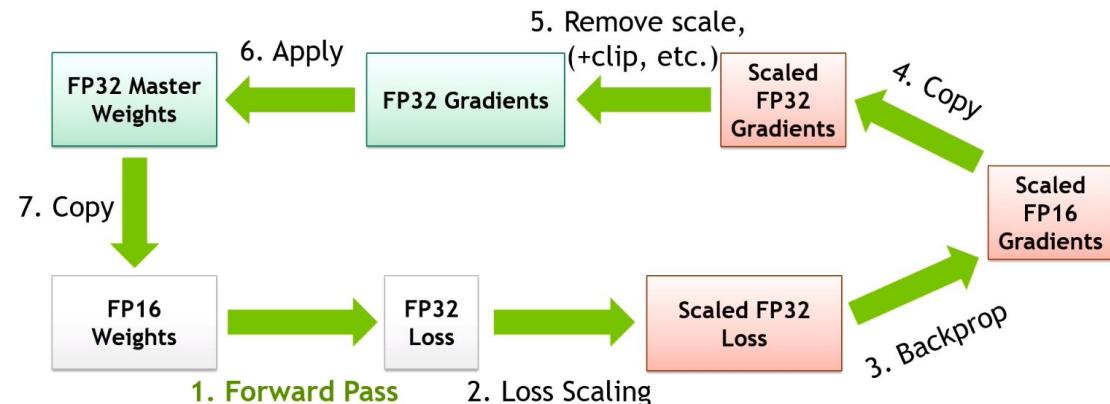
Static memory consumption:

- FP32 master weights and Adam moments (4 bytes x 3 3)
- FP16 copy of weights for calculations (2 bytes)
- FP16 gradients (2 bytes)

7B model = $7 \times 10^9 \times (4 \times 3 + 2 + 2)$
bytes = 112GB

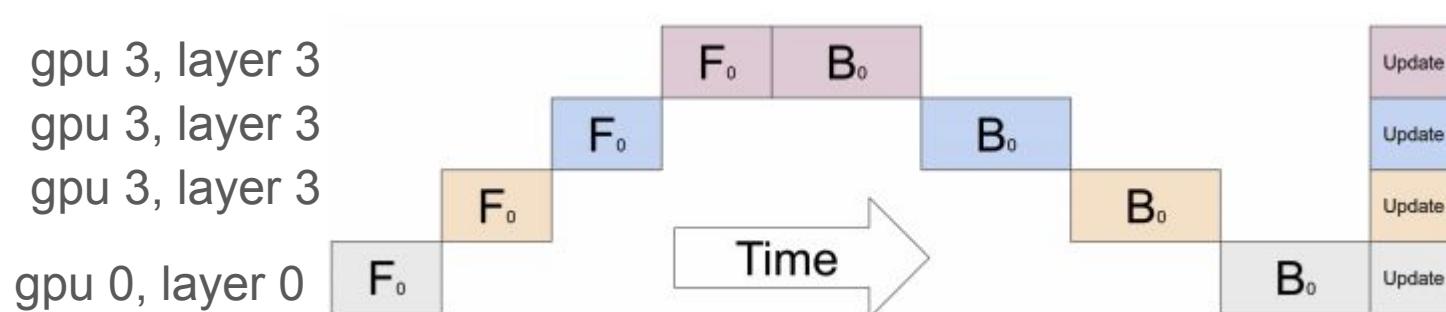
70B model = $70 \times 10^9 \times (4 \times 3 + 2 + 2)$
bytes = 1.1TB

MIXED PRECISION TRAINING



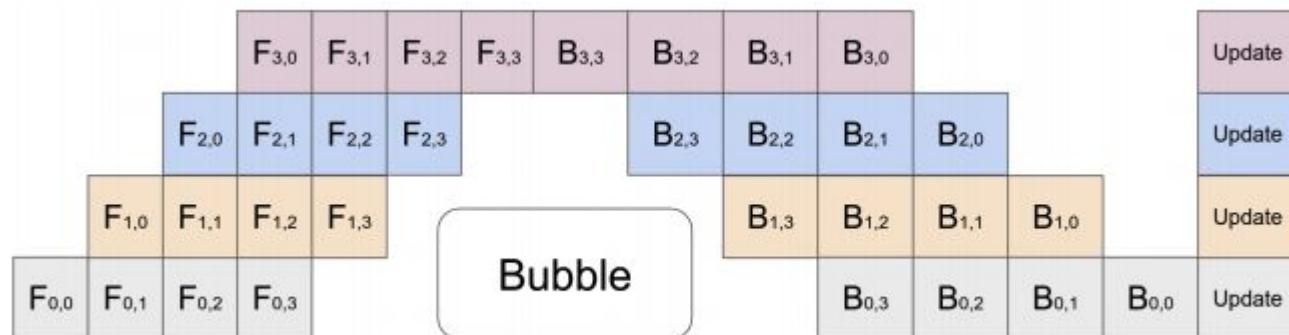
Pipeline parallel

each stage holds contiguous layers



Pipeline parallel

- Micro-batching reduces bubbles
- Pipeline parallel is very good choice when it's possible to use zero bubble strategies



Pipeline parallel

- Bubbles are hard to get rid of
- Requires changing training code

```
with torch.device("meta"):
    assert num_stages == 2, "This is a simple 2-stage example"

    # we construct the entire model, then delete the parts we do not need
    # this stage
    # in practice, this can be done using a helper function that automatically
    # divides up layers across stages.
    model = Transformer()

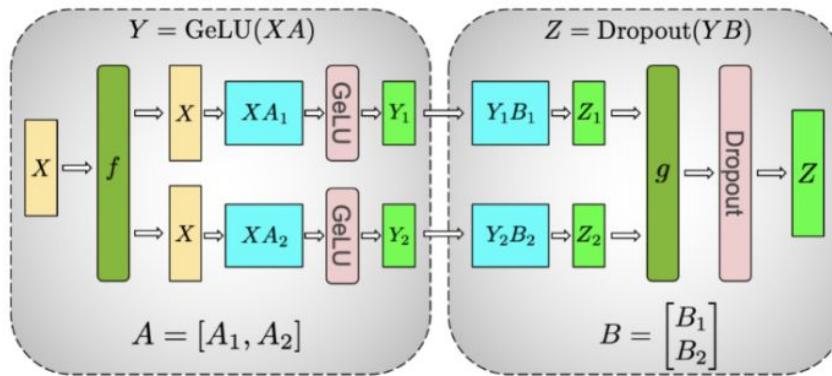
    if stage_index == 0:
        # prepare the first stage model
        del model.layers["1"]
        model.norm = None
        model.output = None

    elif stage_index == 1:
        # prepare the second stage model
        model.tok_embeddings = None
        del model.layers["0"]

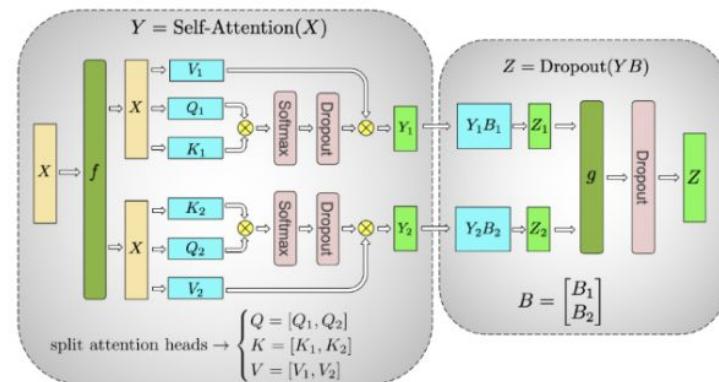
from torch.distributed.pipeline import PipelineStage
stage = PipelineStage(
    model,
    stage_index,
    num_stages,
    device,
    input_args=example_input_microbatch,
)
```

Tensor parallel

Parts of matmul are calculated across several GPUs and then combined



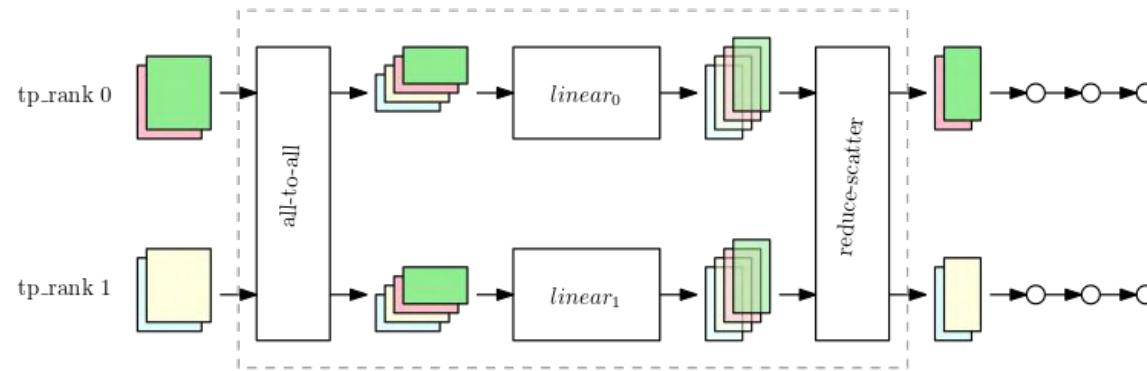
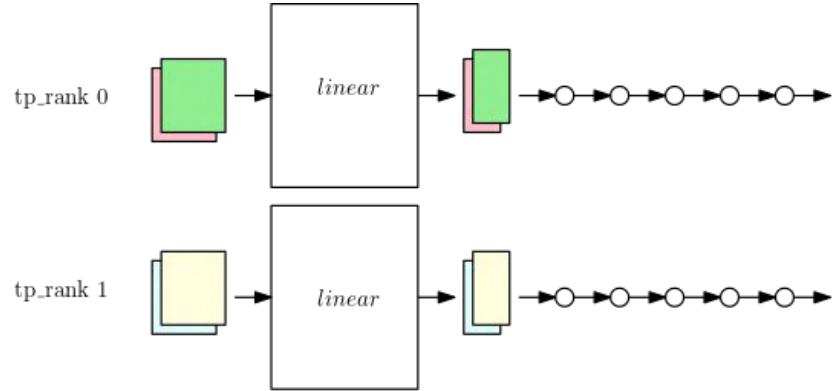
(a) MLP



(b) Self-Attention

Tensor parallel

- Communication across GPUs can't be overlapped with computation



Tensor parallel

- Communication across GPUs can't be overlapped with communication
- (Megatron example with 1.2 rows)
Complicated code changes are required

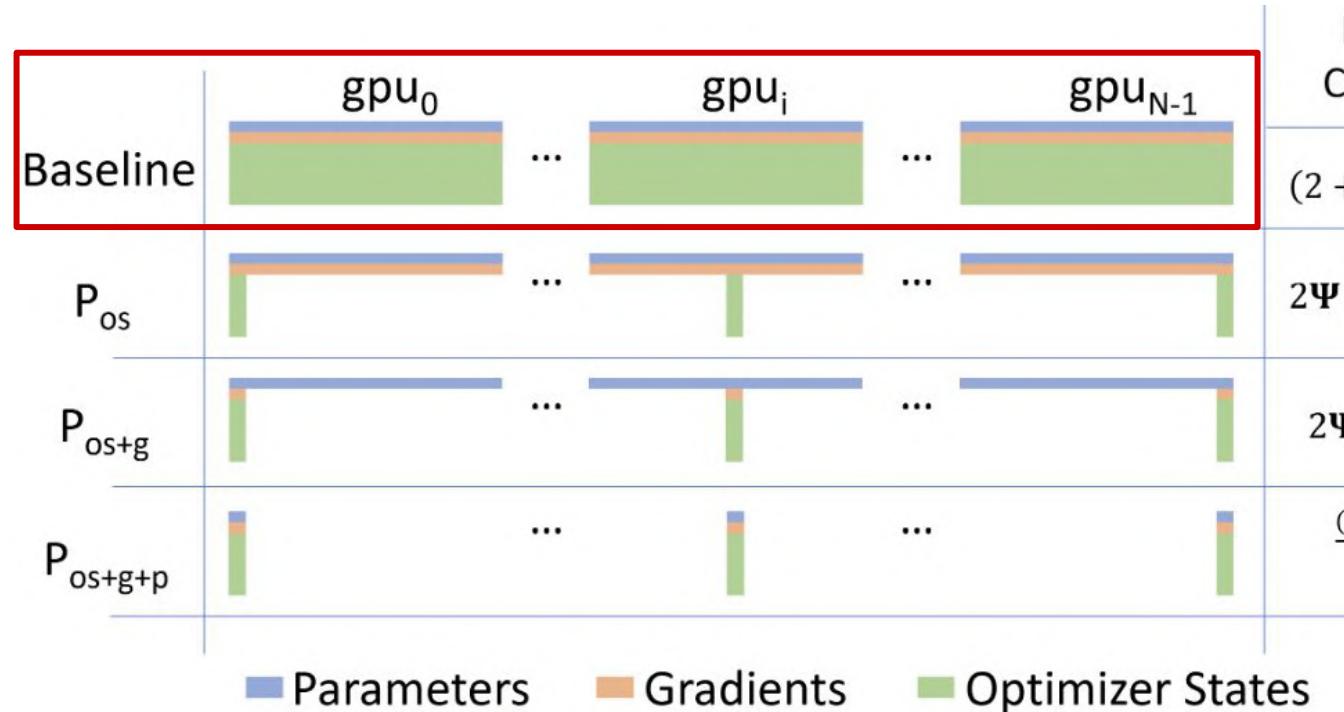
Code Blame 1224 lines (1049 loc) · 46.7 KB

Raw  

```
997 class RowParallelLinear(torch.nn.Module):
1147     def forward(self, input_):
1198         output_bias = None
1199         else:
1200             output = output_
1201             output_bias = self.bias
1202             return output, output_bias
1203
1204     def sharded_state_dict(self, prefix='', sharded_offsets=(), metadata=None):
1205         """Sharding along axis 1, bias not sharded"""
1206         state_dict = self.state_dict(prefix='', keep_vars=True)
1207         return make_sharded_tensors_for_checkpoint(
1208             state_dict, prefix, {'weight': 1}, sharded_offsets
1209         )
1210
1211     def set_extra_state(self, state: Any):
1212         """Extra state is ignored"""
1213
1214     def get_extra_state(self) -> None:
1215         """Keep compatibility with TE state dict."""
1216         return None
1217
1218     def __repr__(self):
1219         tp = self.input_size // self.input_size_per_partition
1220         use_bias = self.bias is not None and self.bias is True
1221         return (
1222             f"{type(self).__name__}({in_features={self.input_size}, "
1223             f"out_features={self.output_size}, bias={use_bias}, TP={tp})"
1224         )
```

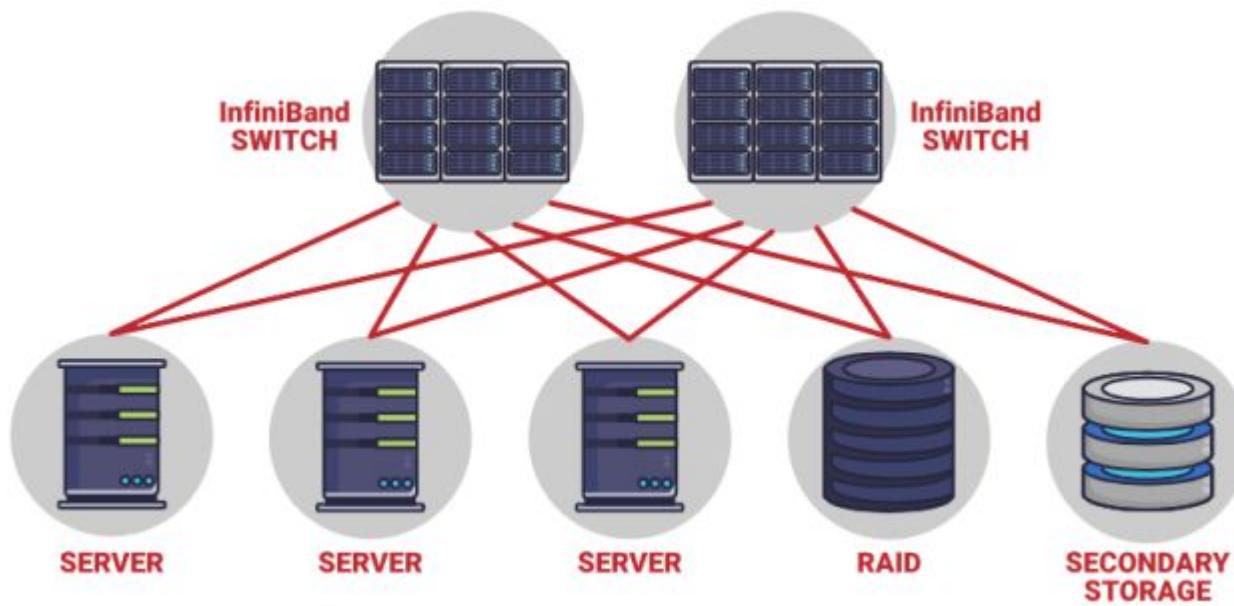
So what?...

Notice how all GPUs store the same values in their memory in DDP

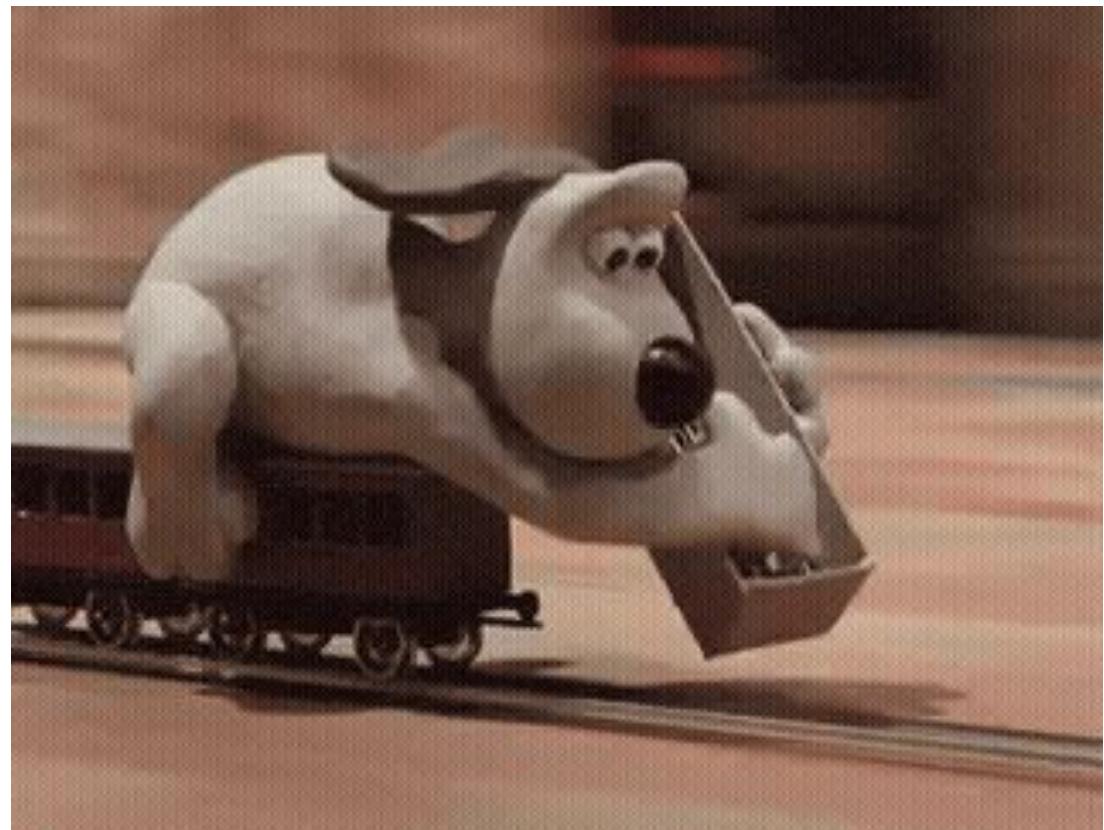


So what?...

Infiniband fabric is idle during forward



Lets load the weights right
before we need them.



Fully Sharded Data Parallel

- Separate model into parts (FSDP unit)

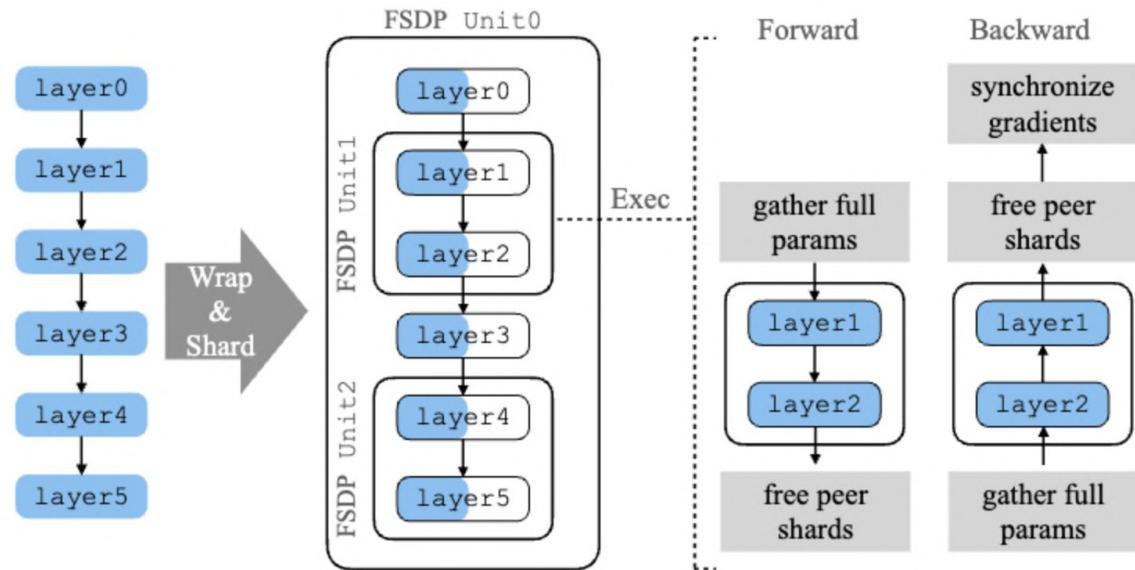


Figure 1: FSDP Algorithm Overview

Fully Sharded Data Parallel

- Separate model into parts (FSDP unit)
- For 32 GPU training each GPU stores:
1/32 of Unit0
1/32 of Unit1
1/32 of Unit2

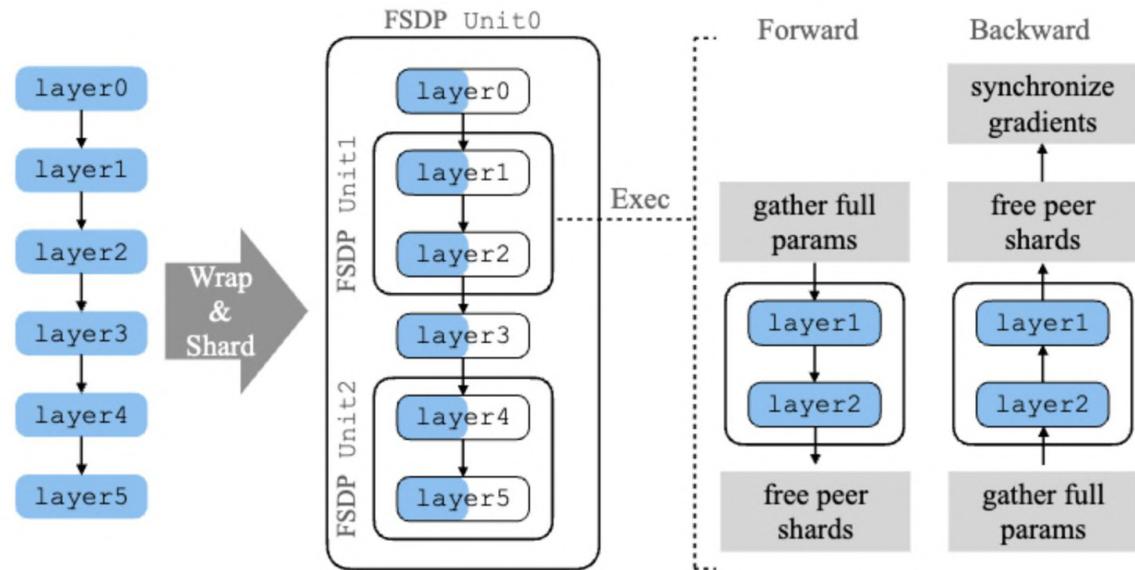


Figure 1: FSDP Algorithm Overview

Fully Sharded Data Parallel

- Separate model into parts (FSDP unit)
- For 32 GPU training each GPU stores 1/32 of every unit.
- On-demand gather **16-bit** parameters for forward or backward

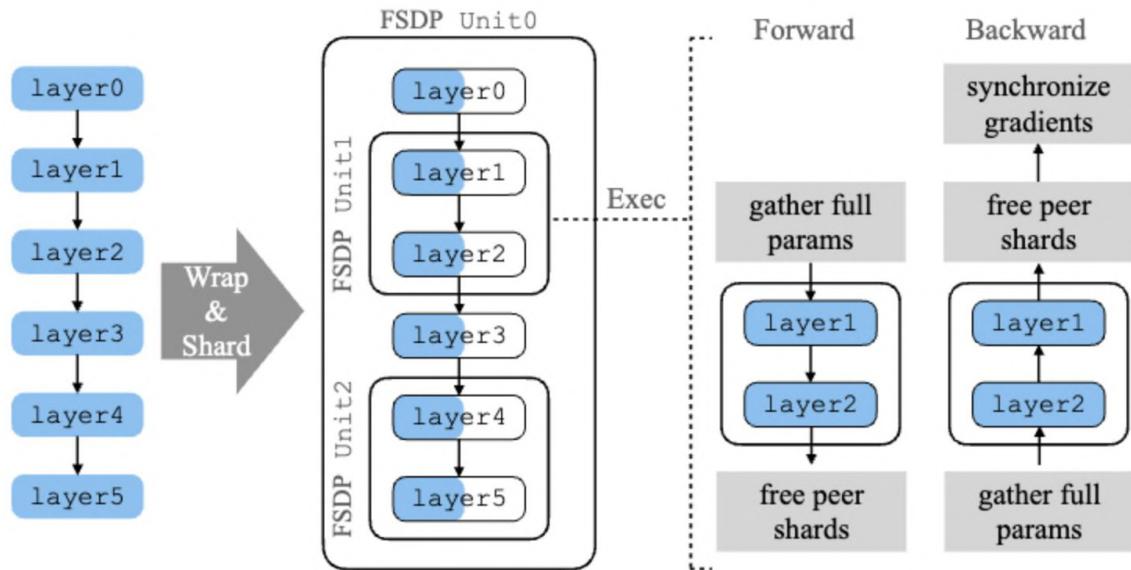


Figure 1: FSDP Algorithm Overview

Fully Sharded Data Parallel

FlatParam has 2 states

- sharded
 - unsharded

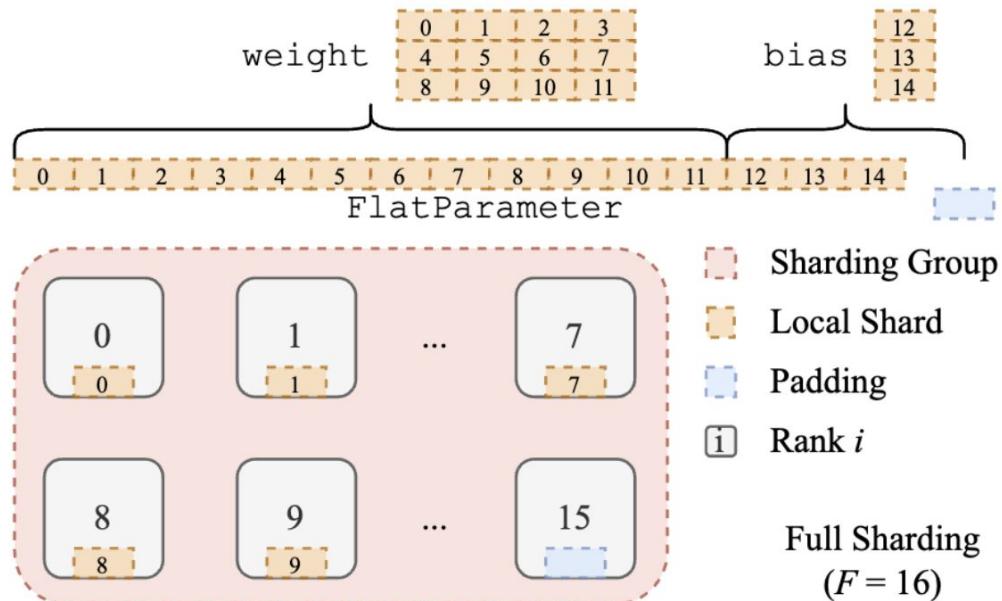
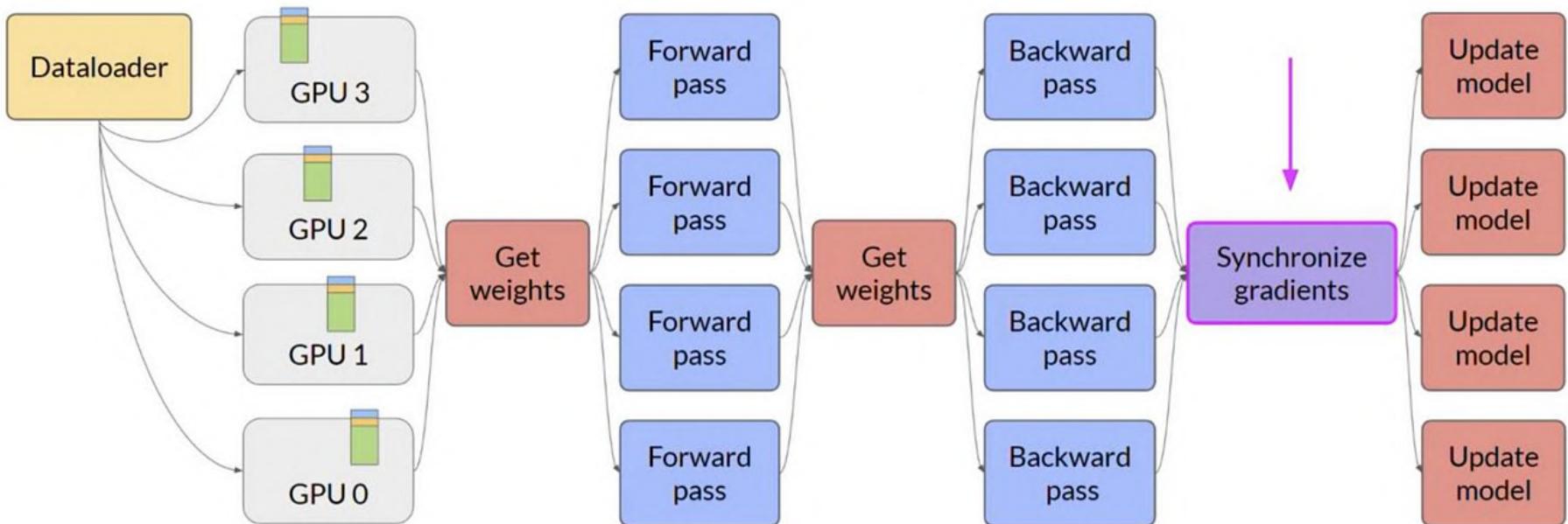


Figure 3: Full Sharding Across 16 GPUs

Fully Sharded Data Parallel (FSDP)



NCCL Crash course

- torch.distributed give us **collective communication primitives**
- torch.distributed can use several backends
- NCCL is most popular backend for GPU training

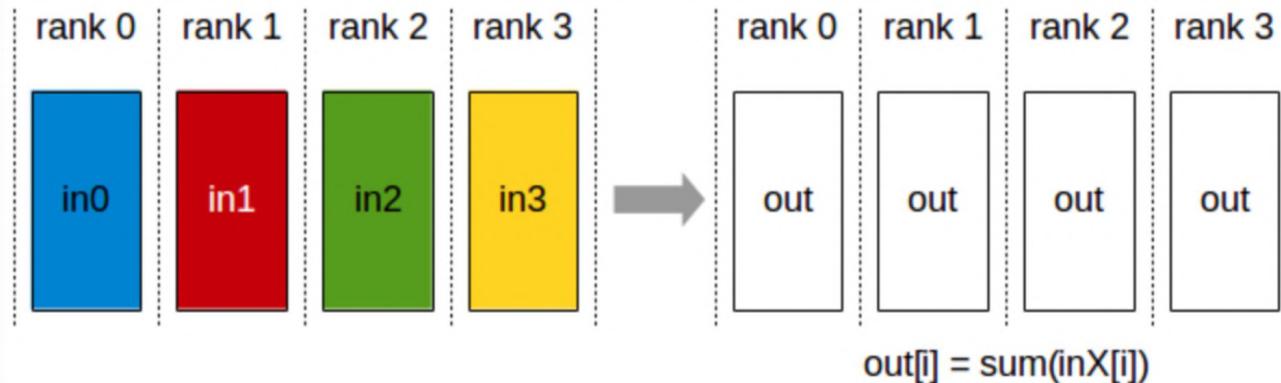
```
torch.distributed.init_process_group(backend=None, init_method=None,
timeout=None, world_size=-1, rank=-1, store=None, group_name=' ',
pg_options=None, device_id=None) [SOURCE]
```

NCCL Crash course

AllReduce

The AllReduce operation performs reductions on data (for example, sum, min, max) across devices and stores the result in the receive buffer of every rank.

In a *sum* allreduce operation between k ranks, each rank will provide an array in of N values, and receive identical results in array out of N values, where $\text{out}[i] = \text{in}0[i] + \text{in}1[i] + \dots + \text{in}(k-1)[i]$.



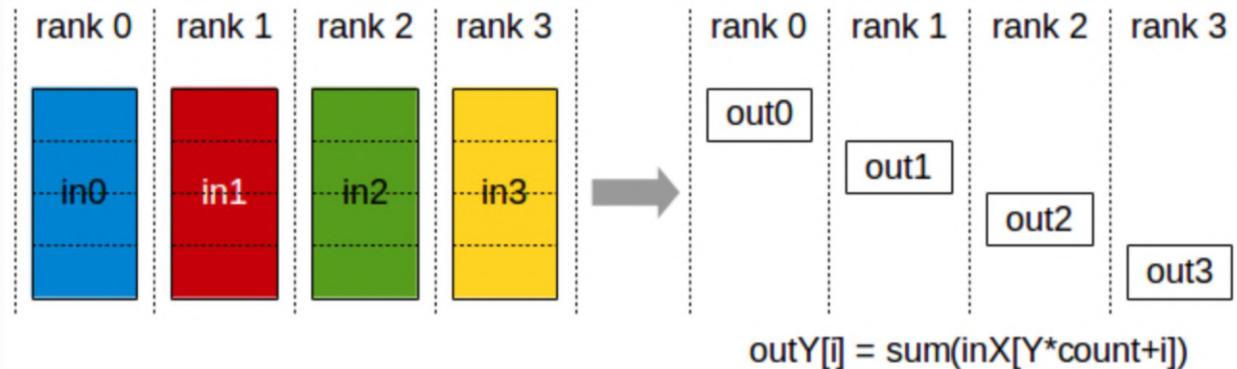
All-Reduce operation: each rank receives the reduction of input values across ranks.

NCCL Crash course

- ## ReduceScatter

The ReduceScatter operation performs the same operation as Reduce, except that the result is scattered in equal-sized blocks between ranks, each rank getting a chunk of data based on its rank index.

The ReduceScatter operation is impacted by a different rank to device mapping since the ranks determine the data layout.

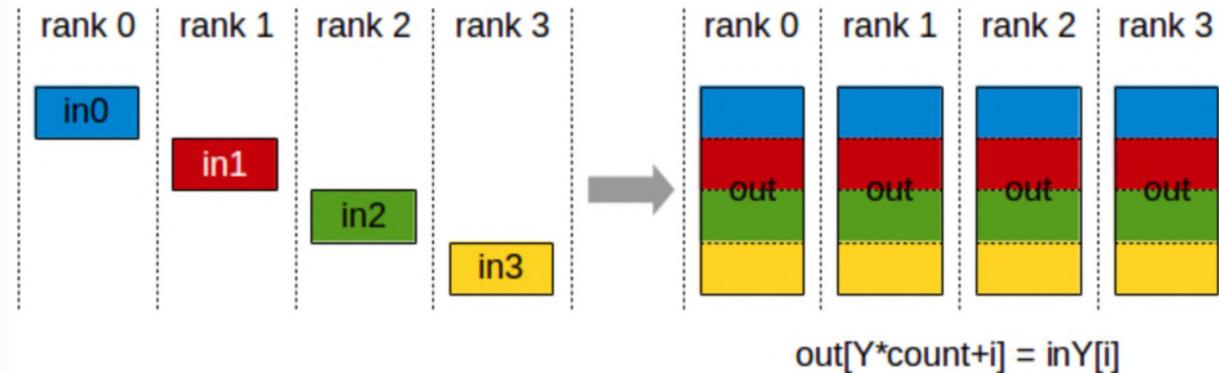


NCCL Crash course

AllGather

The AllGather operation gathers N values from k ranks into an output buffer of size $k \times N$, and distributes that result to all ranks.

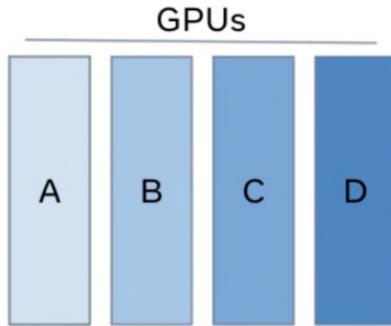
The output is ordered by the rank index. The AllGather operation is therefore impacted by a different rank to device mapping.



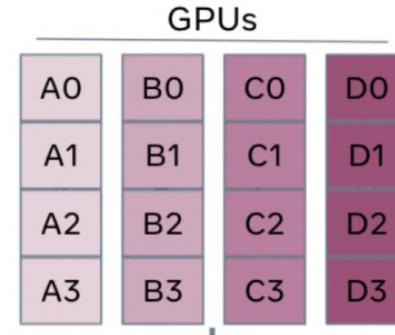
AllGather operation: each rank receives the aggregation of data from all ranks in the order of the ranks.

NCCL Crash course

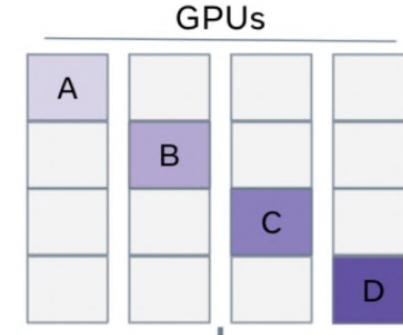
All Reduce



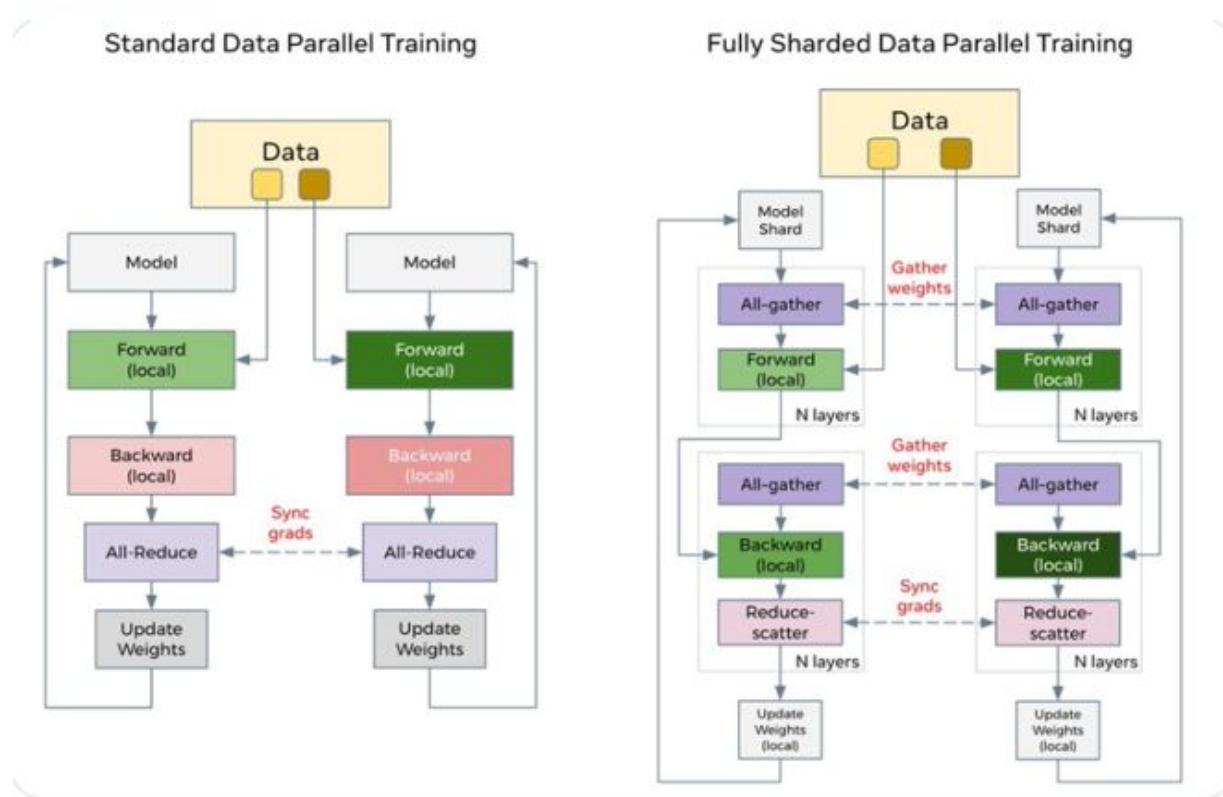
Reduce- Scatter



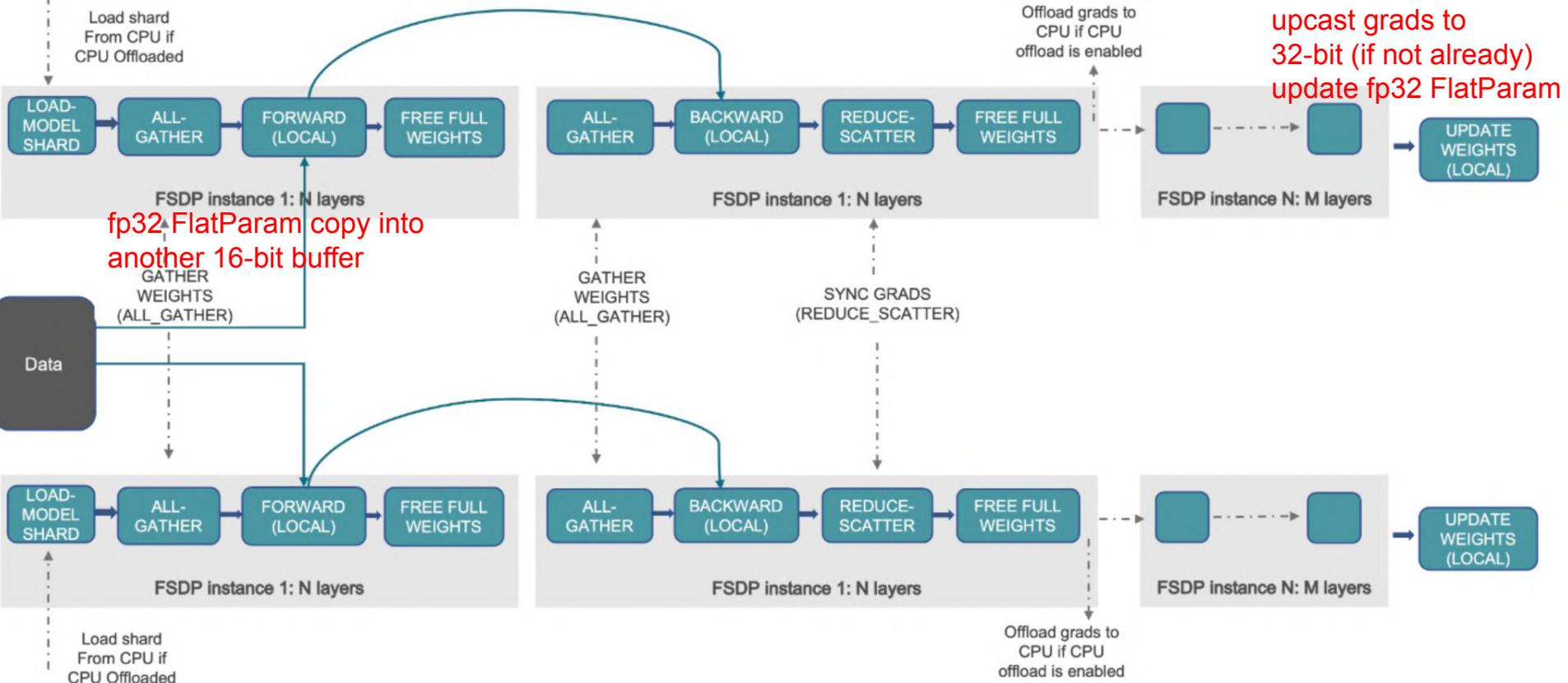
All-gather



FSDP high-level overview



FSDP (More details)



FSDP for Llama 70B on 128 GPUs

FSDP unit: 1 transformer layer (0.9B parameters)

80 FlatParams: $70\text{B} * 4\text{bytes} / 128 = \mathbf{2.1\text{GB}}$

80 Adam's avg of grads: $70\text{B} * 4\text{bytes} / 128 = \mathbf{2.1\text{GB}}$

80 Adam's avg of grads²: $70\text{B} * 4\text{bytes} / 128 = \mathbf{2.1\text{GB}}$

1 buffer for fp16/bf16 weights: $0.9\text{B} * 2\text{bytes} = \mathbf{1.8\text{GB}}$

1 buffer for fp16/bf16 gradients: $0.9\text{B} * 2\text{bytes} = \mathbf{1.8\text{GB}}$

FSDP for Llama 70B on 1280 GPUs

FSDP unit: 1 transformer layer (0.9B parameters)

80 FlatParams: $70\text{B} * 4\text{bytes} / 1280 = \text{Basically 0}$

80 Adam's avg of grads: $70\text{B} * 4\text{bytes} / 1280 = \text{Almost 0}$

80 Adam's avg of grads²: $70\text{B} * 4\text{bytes} / 1280 = \text{Almost 0}$

1 buffer for fp16/bf16 weights: $0.9\text{B} * 2\text{bytes} = \textbf{1.8GB}$

1 buffer for fp16/bf16 gradients: $0.9\text{B} * 2\text{bytes} = \textbf{1.8GB}$

FSDP for Llama 70B on 128 GPUs

At a high level FSDP works as follow:

In constructor

- Shard model parameters and each rank only keeps its own shard

In forward path

- Run all_gather to collect all shards from all ranks to recover the full parameter in this FSDP unit
- Run forward computation
- Discard parameter shards it has just collected

In backward path

- Run all_gather to collect all shards from all ranks to recover the full parameter in this FSDP unit
- Run backward computation
- Run reduce_scatter to sync gradients
- Discard parameters.

FSDP is “almost” free

5. Using tracing functionality

Tracing Crash Course

Model=resnet18

bs=1024

Profiling results can be outputted as a `.json` trace file: Tracing CUDA or XPU kernels Users could switch between cpu, cuda and xpu

```
device = 'cuda'

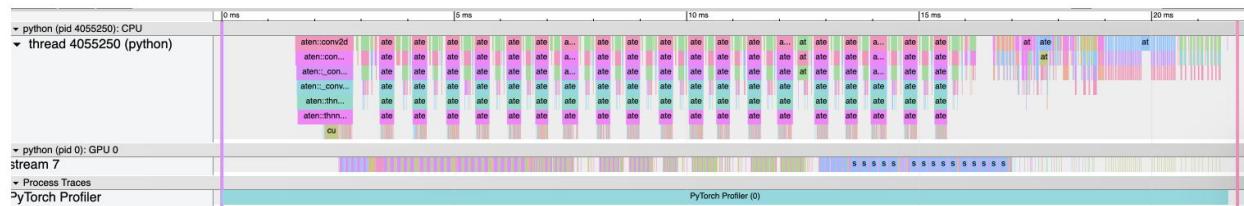
activities = [ProfilerActivity.CPU, ProfilerActivity.CUDA,
    ProfilerActivity.XPU]

model = models.resnet18().to(device)
inputs = torch.randn(5, 3, 224, 224).to(device)

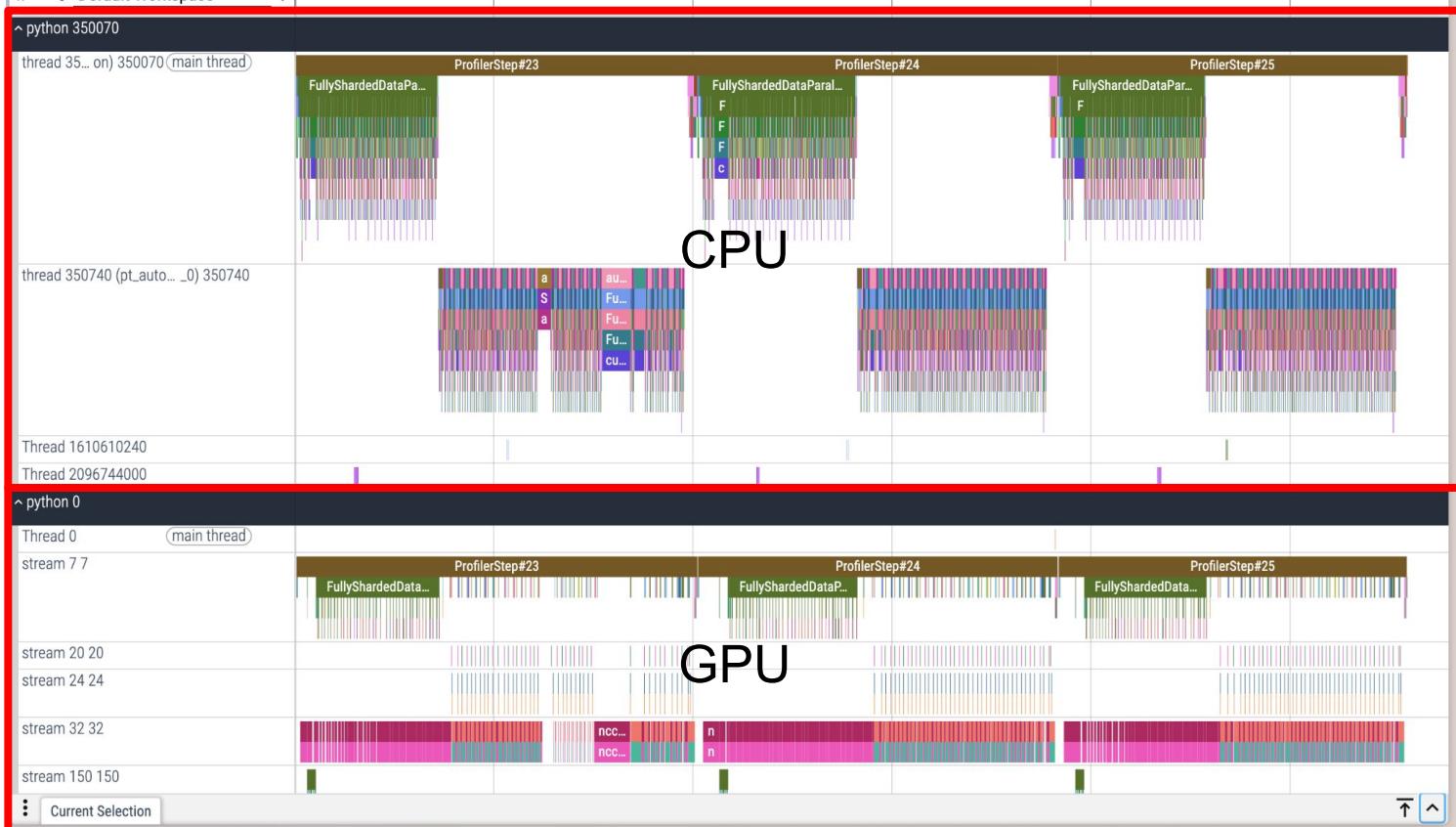
with profile(activities=activities) as prof:
    model(inputs)

prof.export_chrome_trace("trace.json")
```

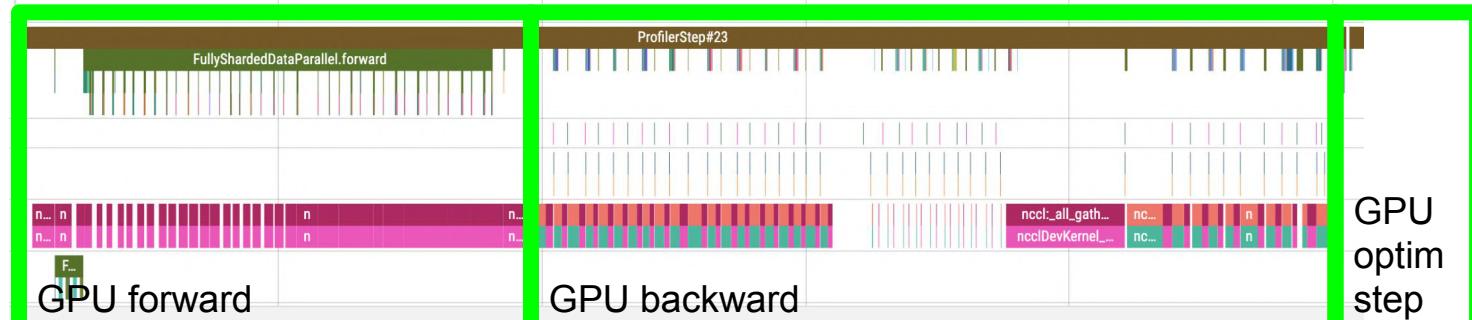
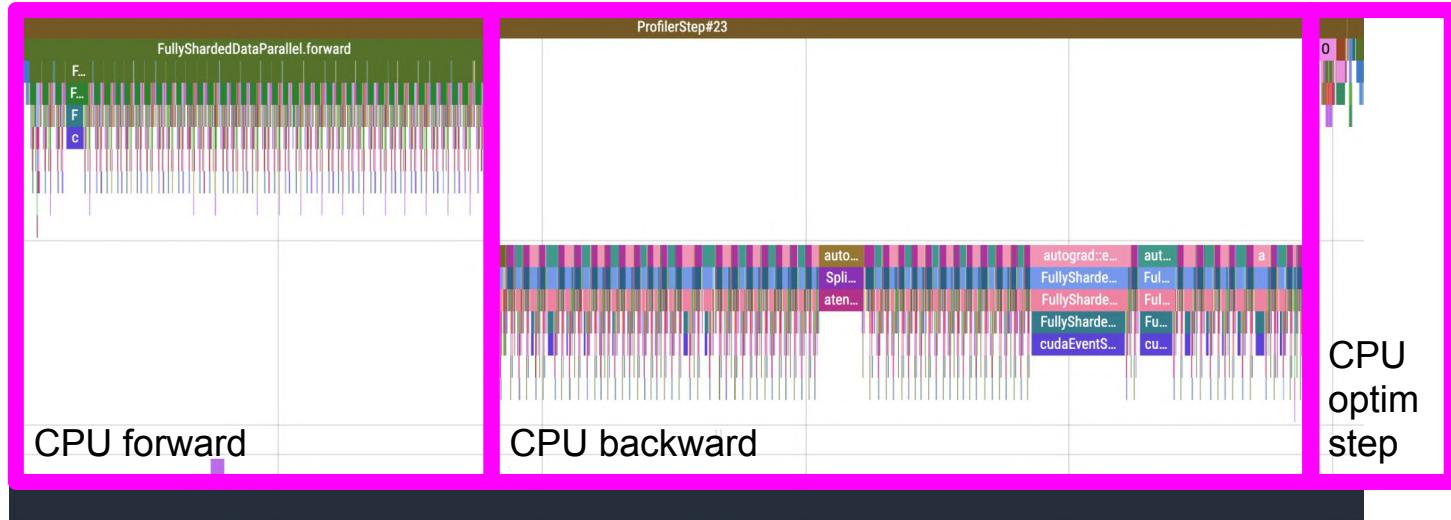
You can examine the sequence of profiled operators and CUDA/XPU kernels in Chrome trace viewer (`chrome://tracing`):



Tracing Crash Course



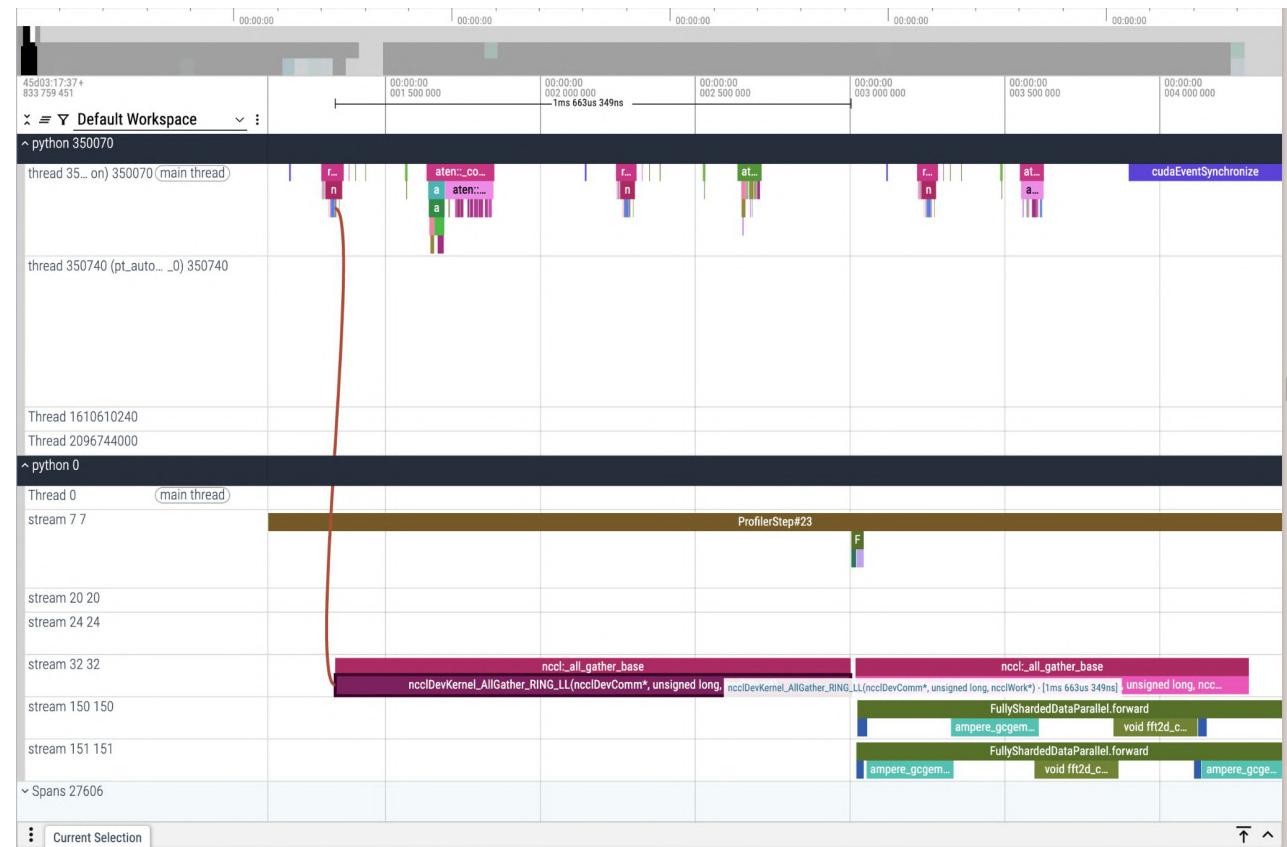
Tracing Crash Course



Tracing Crash Course

CPU issues

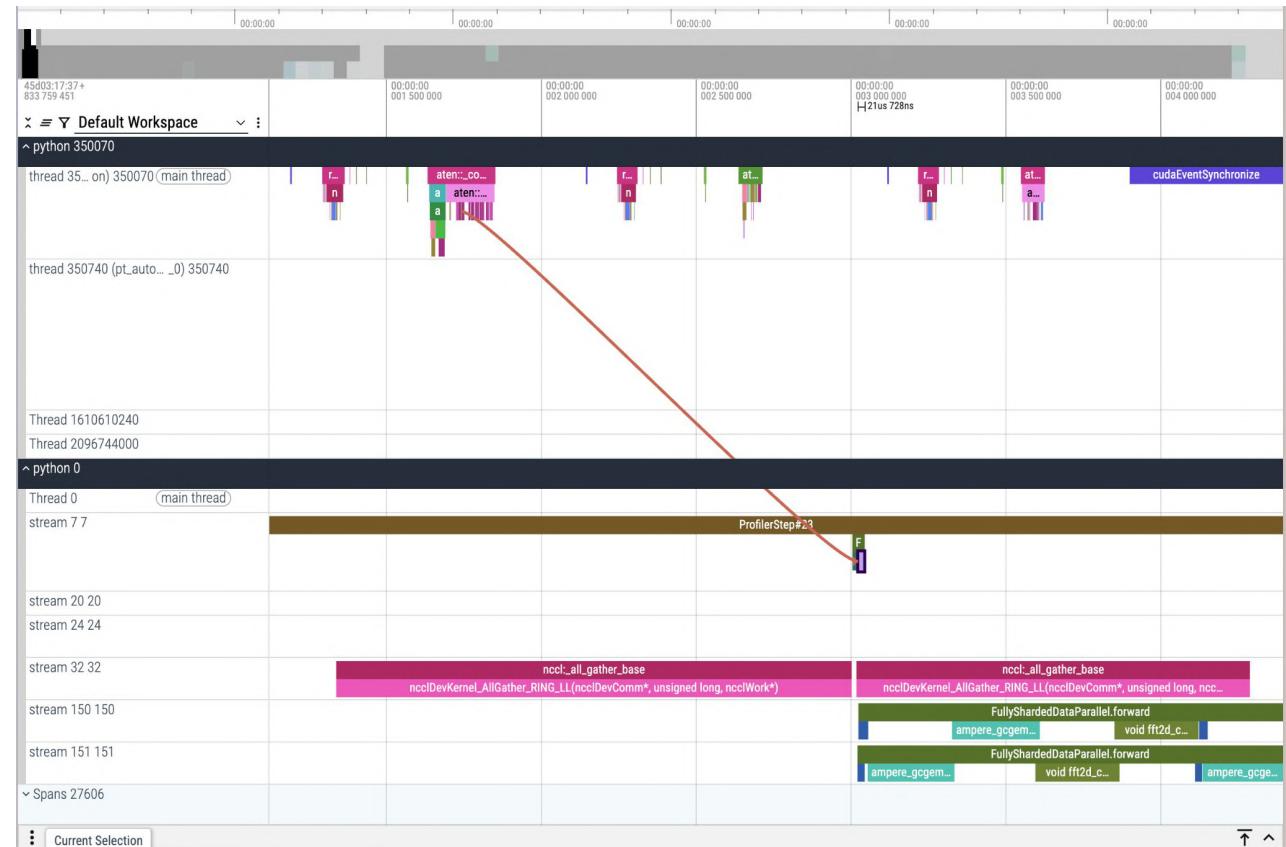
AllGather kernel into stream 32



Tracing Crash Course

CPU synchronizes stream 7 with stream 32
(stream 7 will wait until all currently scheduled tasks in stream 32 end)

CPU issues convolution kernel in stream 7



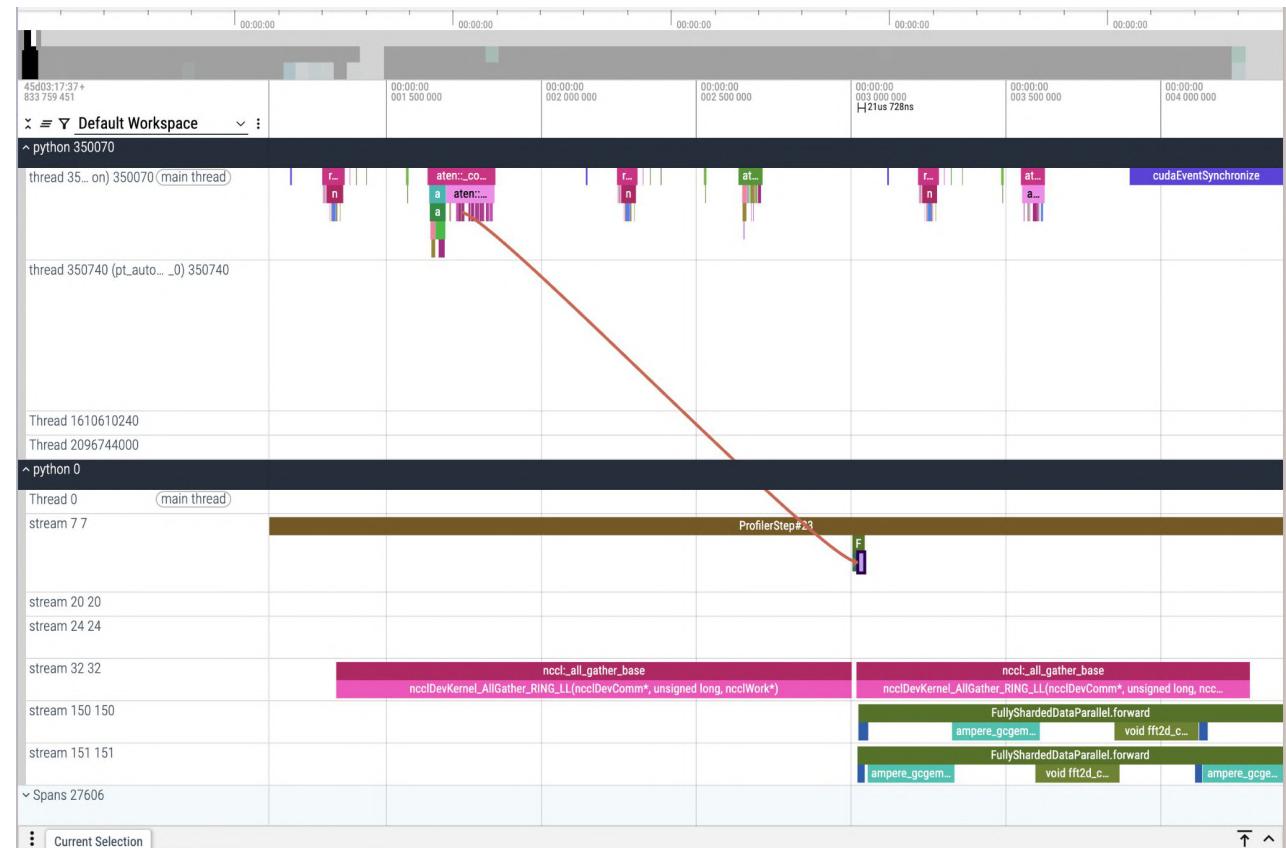
Tracing Crash Course

CUDA streams are just different **queues** of kernels.

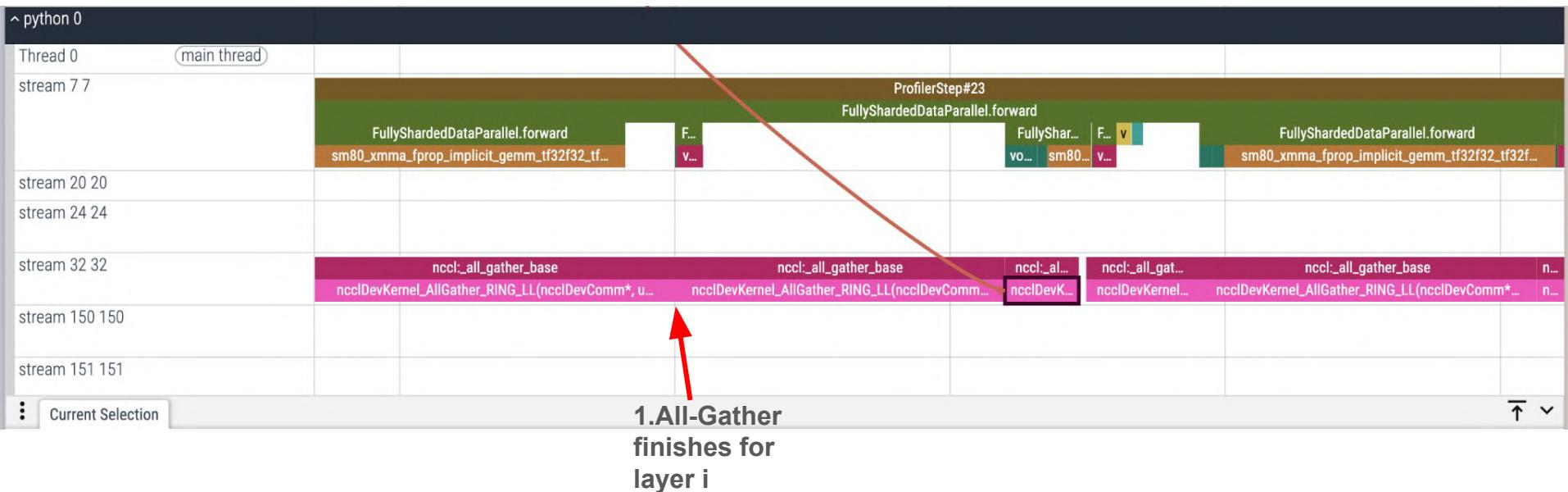
Kernels submitted to one stream always execute sequentially.

More on the matter:

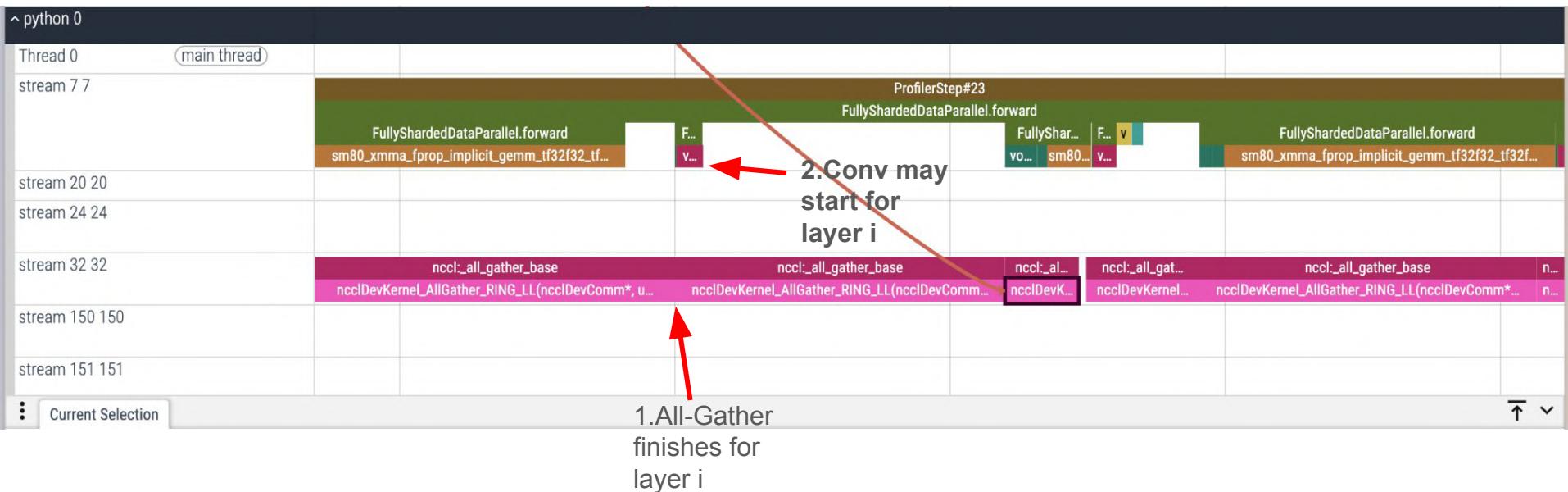
<https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf>



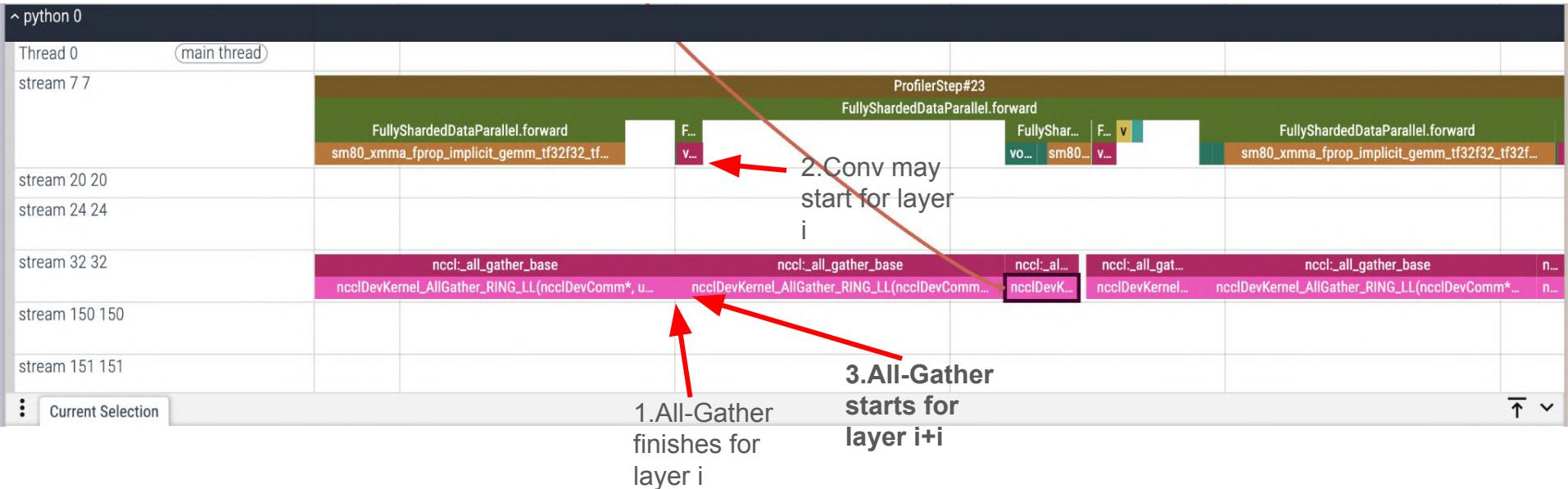
Tracing Crash Course



Tracing Crash Course

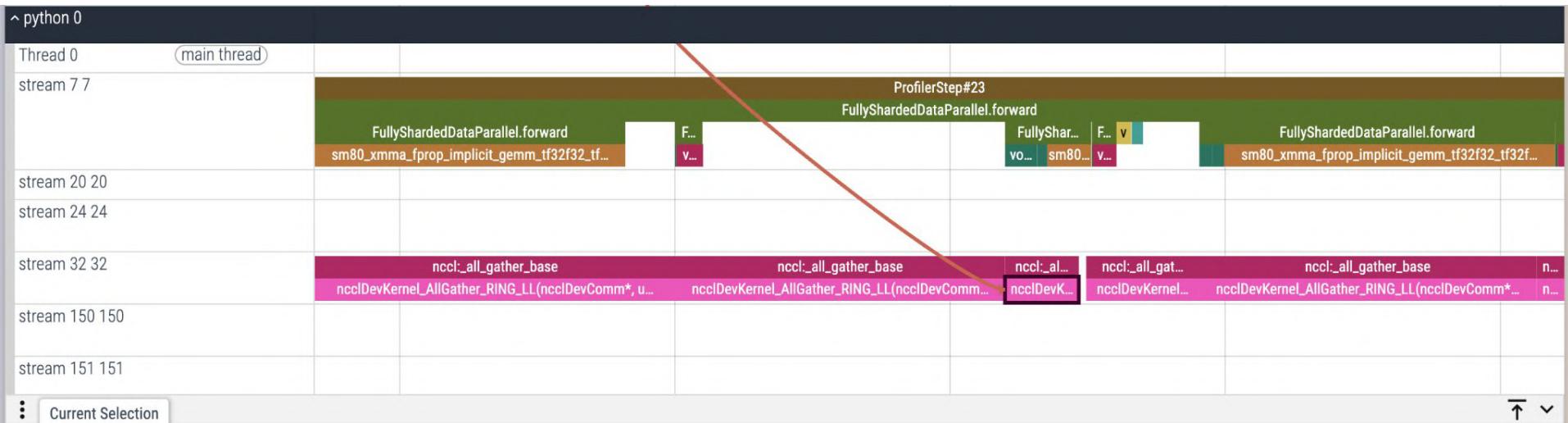


Tracing Crash Course

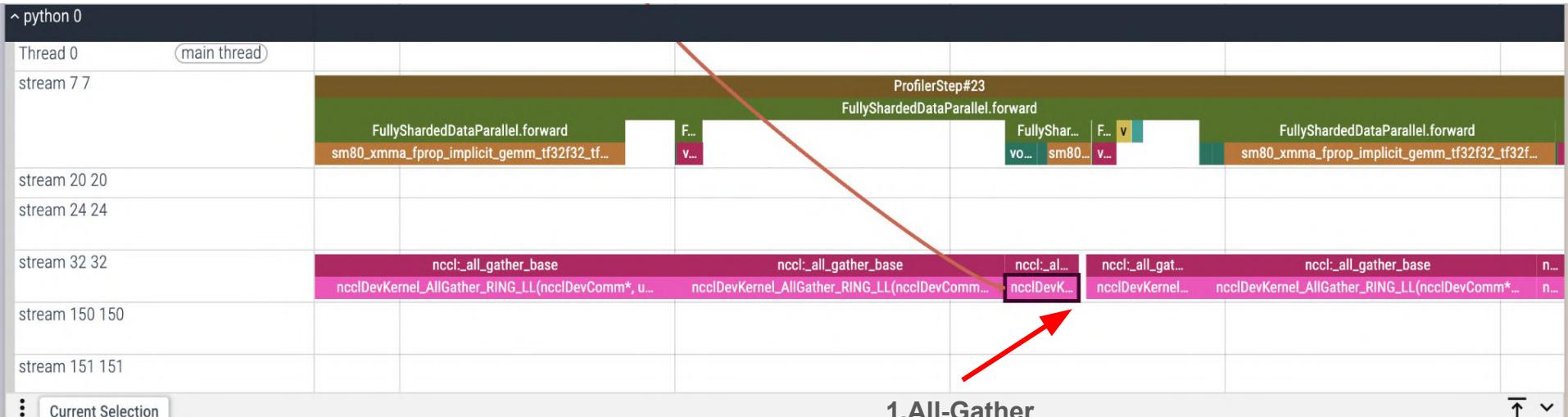


Tracing Crash Course

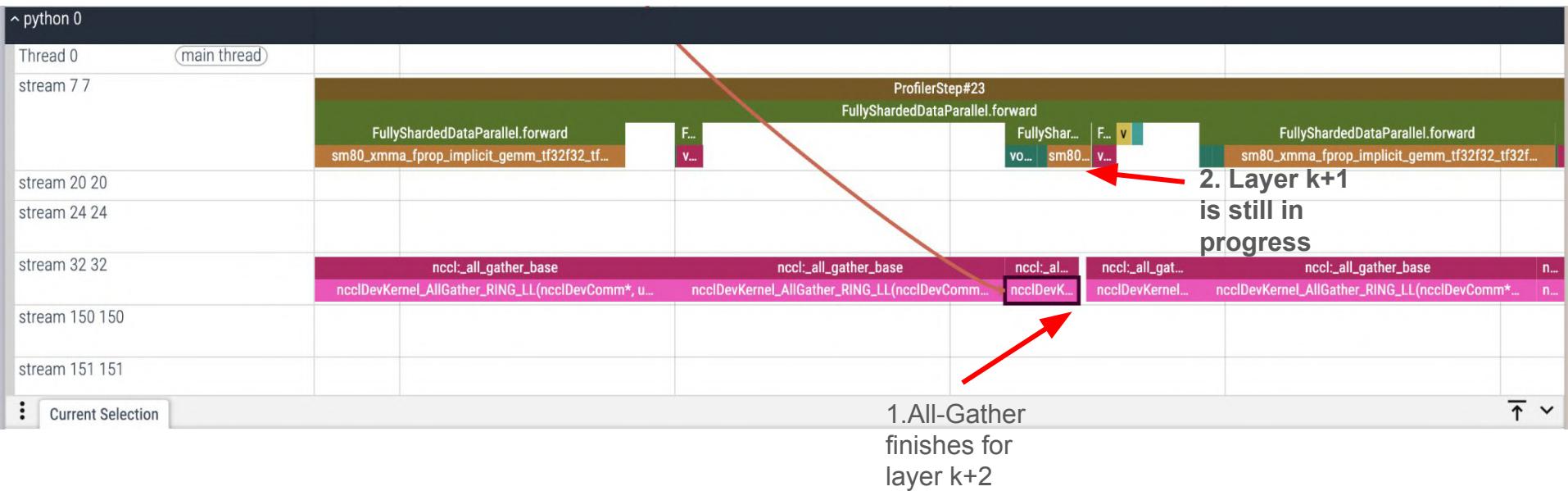
Communication is longer than computation!
GPU IS IDLE!!!



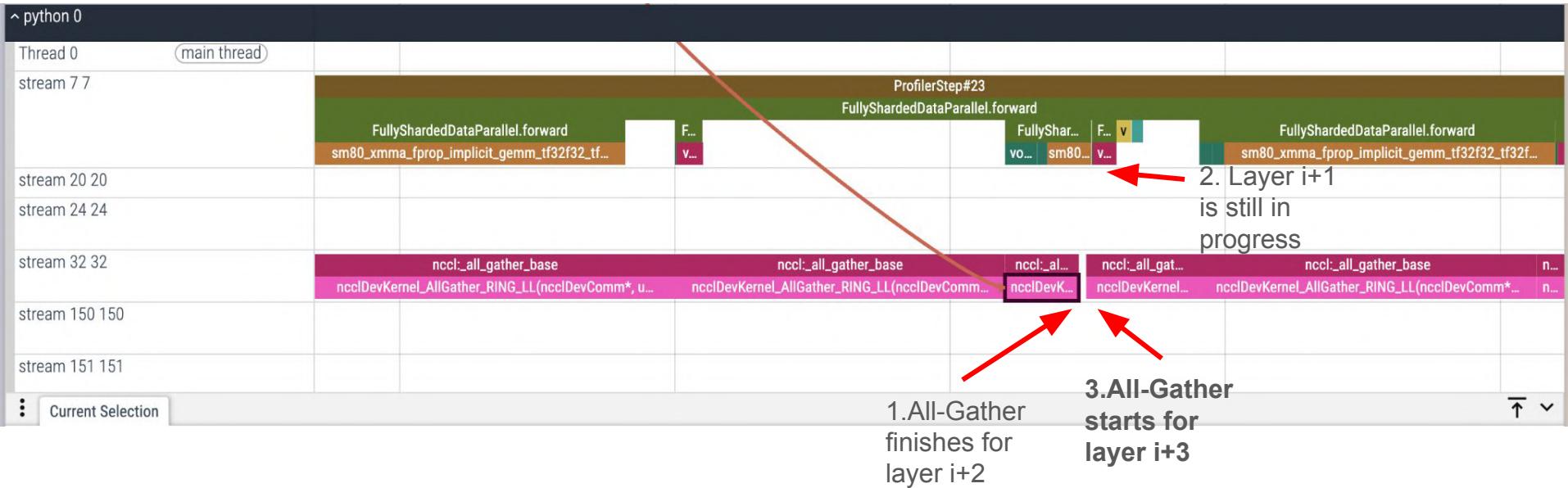
Tracing Crash Course



Tracing Crash Course



Tracing Crash Course



Tracing Crash Course

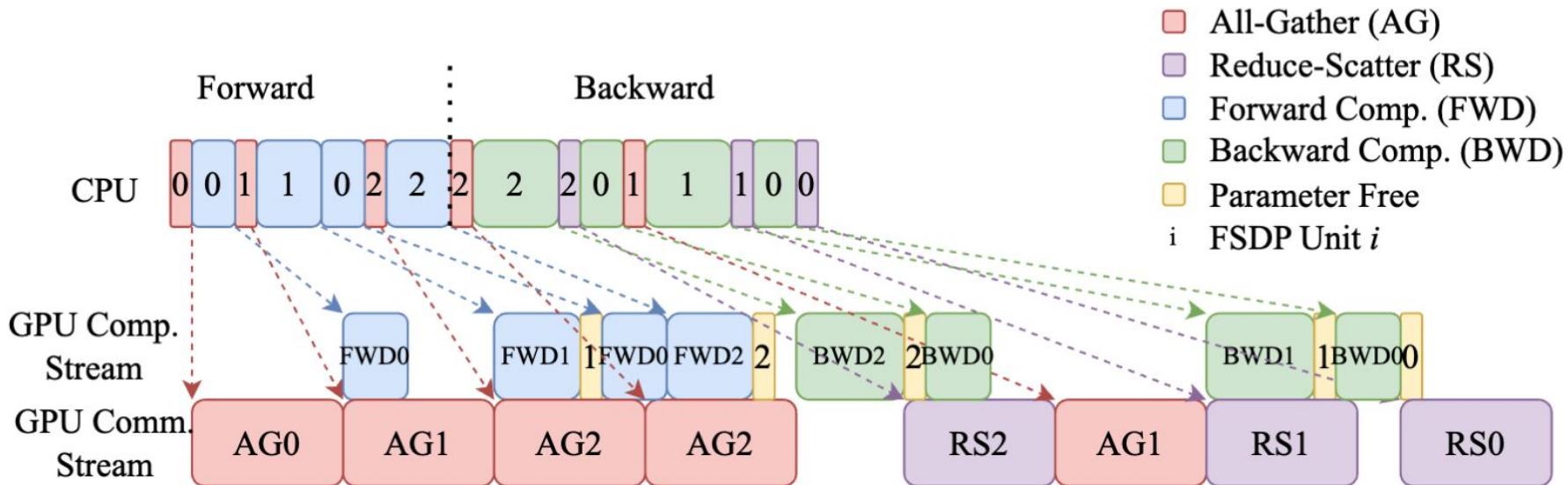


Figure 5: Overlap Communication and Computation

Tracing Crash Course

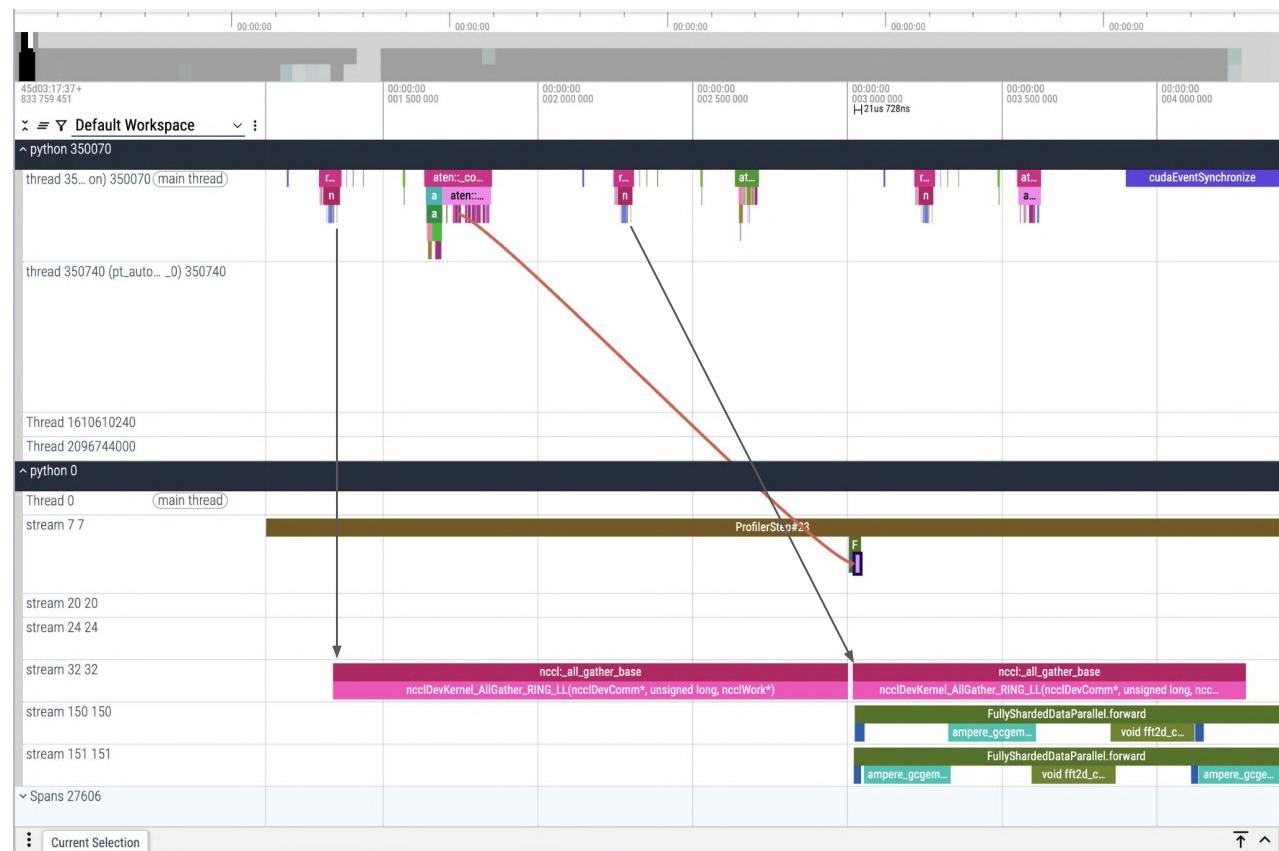
- FSDP allows full **computation/communication overlap** if **All-Gather** and **Reduce-Scatter** are fast enough
- Communication speed can be influenced by
 - Hardware speed
 - Cluster topology
 - NCCL version
 - Model sharding strategy (We will focus on this one!)

Prefetching

Implicit forward prefetching

CPU thread runs “in front” of GPU streams

All-Gather starts as soon as it can



Break implicit prefetching

Implicit prefetching breaks if CPU thread waits for GPU

Simplest way to cause CPU-GPU sync is to copy from GPU to CPU

```
def _forward_impl(self, x: Tensor) -> Tensor:  
    # See note [TorchScript super()]  
    x = self.conv1(x)  
    x.cpu()  
    x = self.bn1(x)  
    x.cpu()  
    x = self.relu(x)  
    x.cpu()  
    x = self.maxpool(x)  
    x.cpu()  
  
    x = self.layer1(x)  
    x.cpu()  
    x = self.layer2(x)  
    x.cpu()  
    x = self.layer3(x)  
    x.cpu()  
    x = self.layer4(x)  
    x.cpu()
```

Explicit forward prefetching

```
model = FSDP(model, auto_wrap_policy=my_auto_wrap_policy, forward_prefetch=True)
```

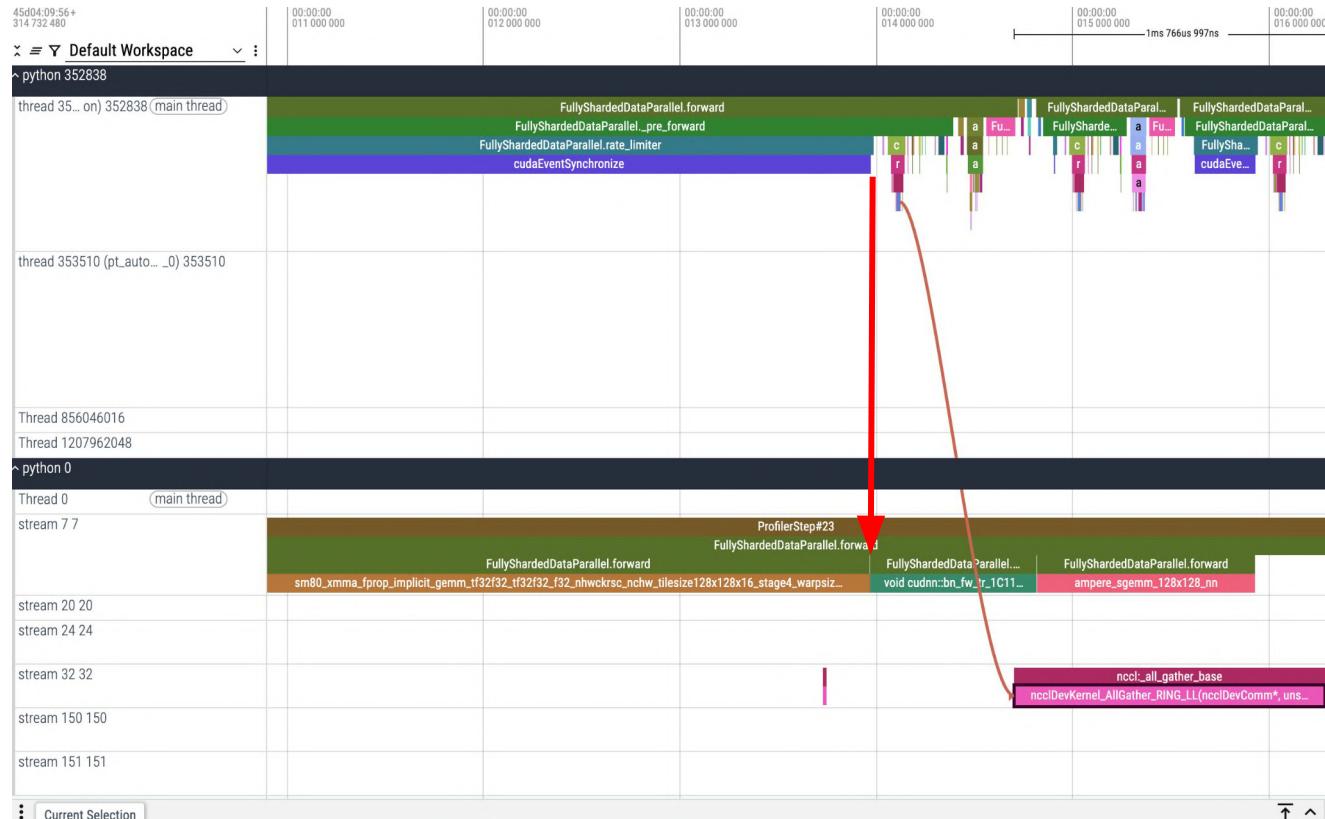
- On first forward records order of All-Gathers
- Later, submits in pre-forward of Module All-Gather for next module too

CPU syncs help limit memory consumption

Limit-all-gather=True

Limit number of
All-Gathers in flight.

Each All-Gather
requires memory.



Explicit backward prefetching

Must use explicit `backward` prefetching or else there will be 0 overlap of communication and computation.

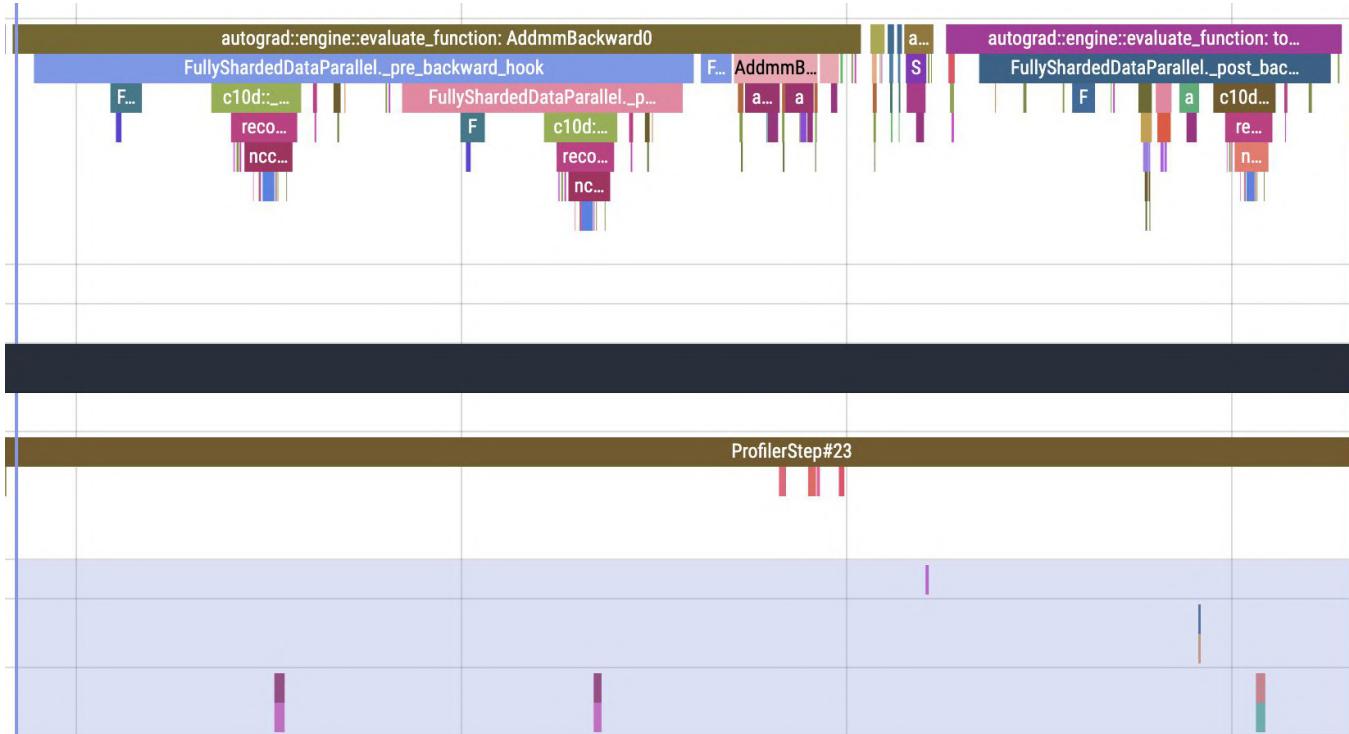
All-Gather i -> Reduce-Scatter i-1 ->All-Gather i+1 -> Reduce-Scatter i

Backward i

-> Backward i+1

Explicit backward prefetching

First pre-backward submits 2 All-Gathers



FSDP sharding levels

From <https://pytorch.org/docs/stable/fsdp.html#torch.distributed.fsdp.ShardingStrategy>

- NO_SHARD (common DDP)
- SHARD_GRAD_OP (no free 16bit-weights after forward)
- FULL_SHARD (was explained in the lecture)
- HYBRID_SHARD

NO_SHARD

Recall:

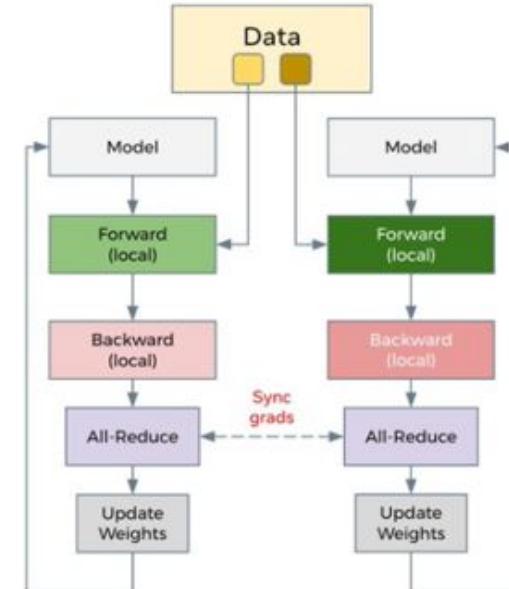
All-Reduce=

Reduce-Scatter + All-Gather

Calculation:

Bytes transmitted in All-reduce =
sizeof(16-bit grads) + sizeof(16-bit grads)

Standard Data Parallel Training

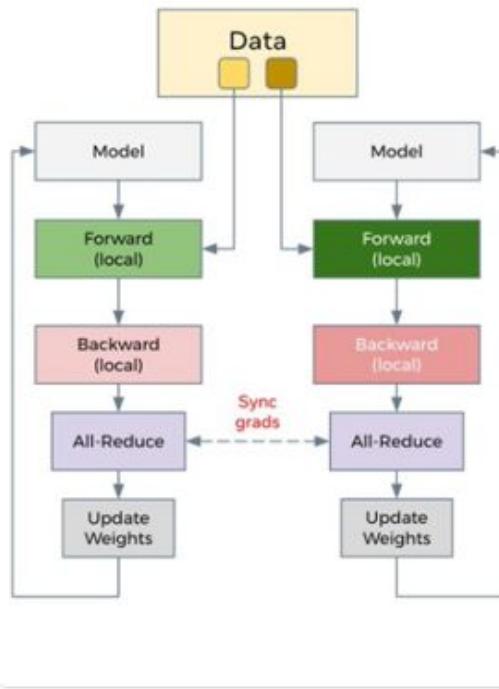


SHARD_GRAD_OP

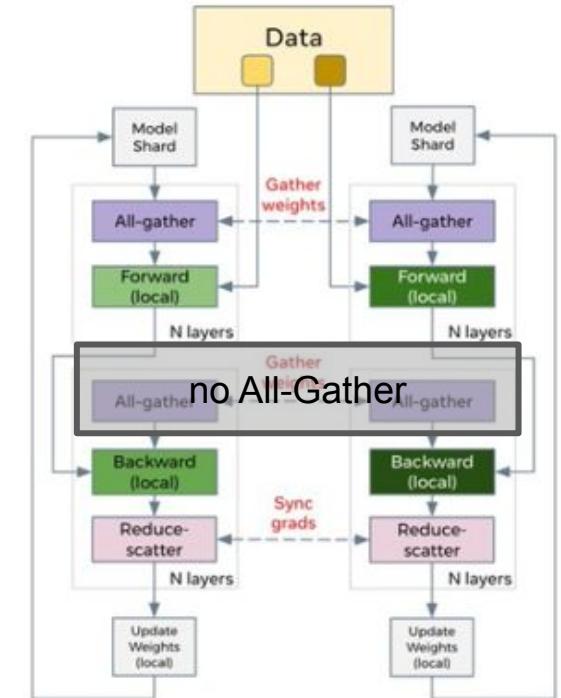
Per forward of FSDP module

**1 All-Gather +
1 Reduce-Scatter**

Standard Data Parallel Training



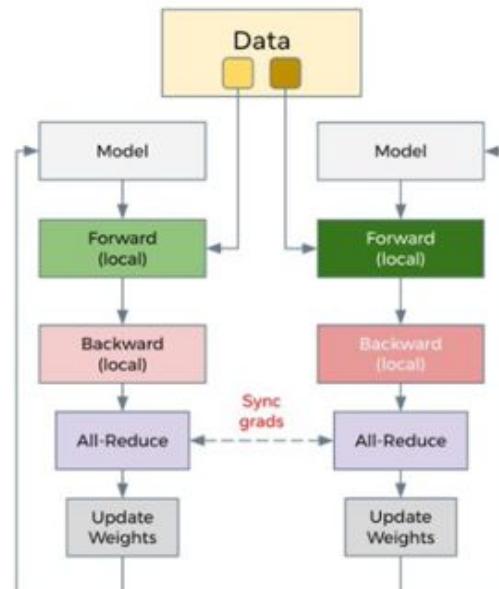
Fully Sharded Data Parallel Training



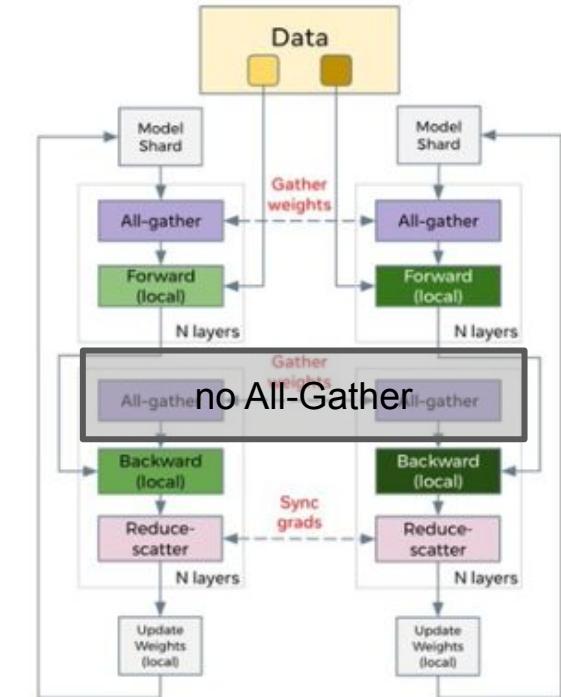
SHARD_GRAD_OP

Theoretically, no communication overhead over DDP!

Standard Data Parallel Training



Fully Sharded Data Parallel Training

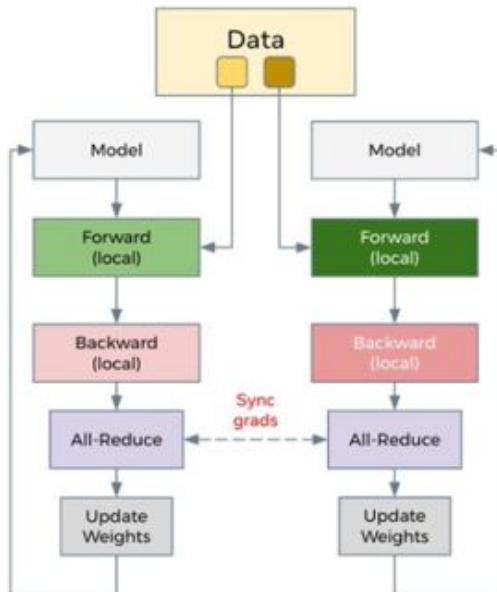


FULL_SHARD

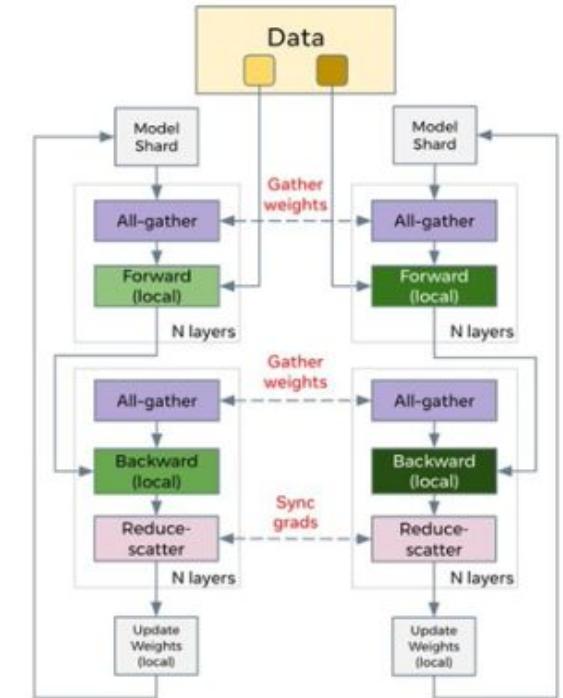
1 Additional
All-Gather

x1.5 communicated
bytes of NO_SHARD

Standard Data Parallel Training



Fully Sharded Data Parallel Training



Deepspeed Crash Course

Deepspeed ZeRO is first implementation of weight sharding

- Shards every nn.Parameter as its own module
- (personal opinion) really hard to understand code and weird API
- ZeRO stage 0 = NO_GRAD
- ZeRO stage 1 = ...
- ZeRO stage 2 = SHARD_GRAD_OP
- ZeRO stage 3 = FULL_SHARD

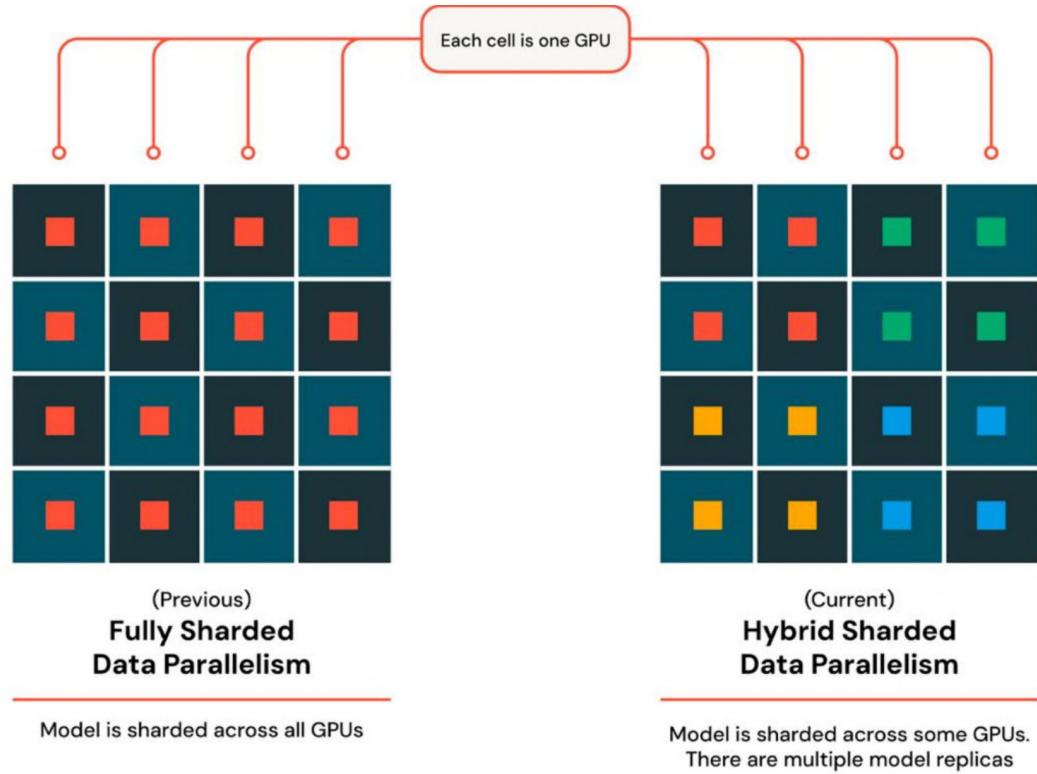
Deepspeed Crash Course

	gpu ₀ ... gpu _i ... gpu _{N-1}	Memory Consumption		Comm Volume
		Formulation	Specific Example $K=12 \Psi=7.5B N_d=64$	
		$(2 + 2 + K) * \Psi$	120GB	
Baseline		$(2 + 2 + K) * \Psi$	120GB	1x
ZeRO1 P_{os}		$2\Psi + 2\Psi + \frac{K * \Psi}{N_d}$	31.4GB	1x
ZeRO2 P_{os+g}		$2\Psi + \frac{(2+K)*\Psi}{N_d}$	16.6GB	1x
ZeRO3 P_{os+g+p}		$\frac{(2 + 2 + K) * \Psi}{N_d}$	1.9GB	1.5x

Legend: Parameters (blue), Gradients (orange), Optimizer States (green)

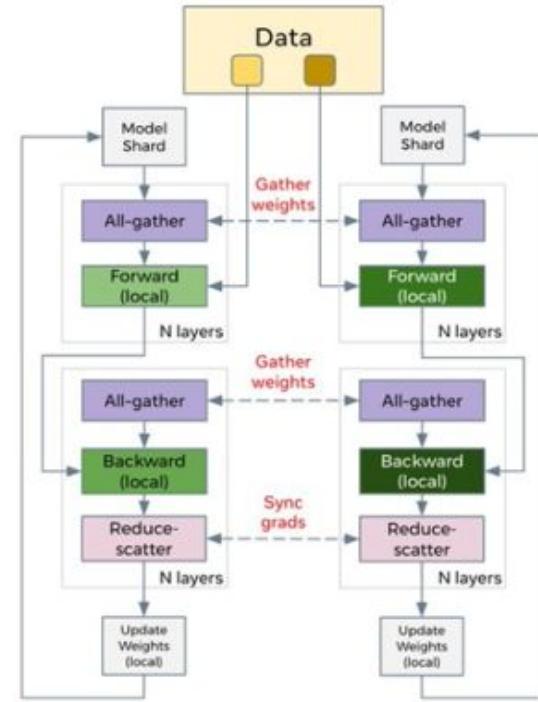
HYBRID_SHARD

Hybrid is between
NO_SHARD and
FULL_SHARD.



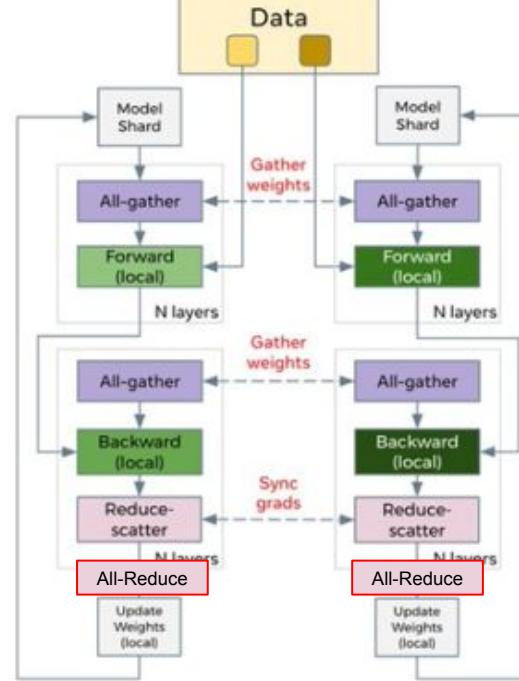
HYBRID_SHARD

Fully Sharded Data Parallel Training



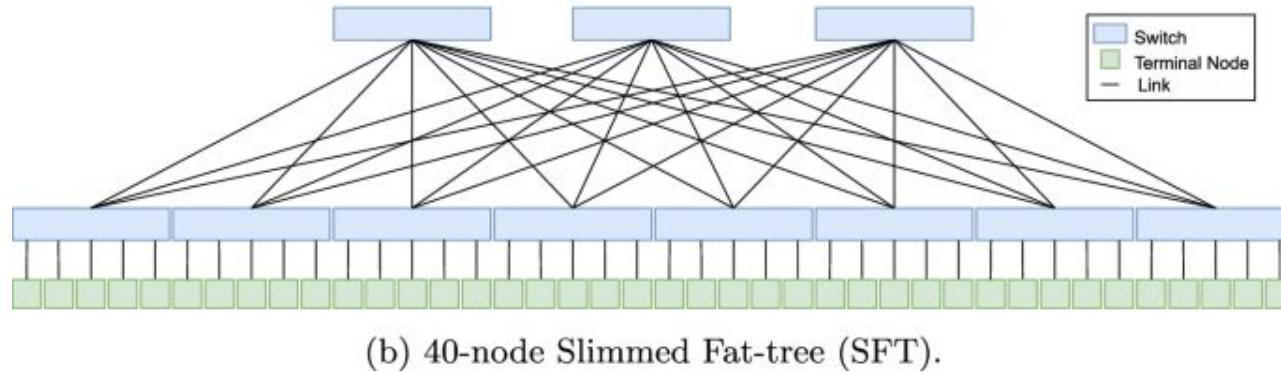
Fully Sharded Data Parallel Training

HYBRID_SHARD



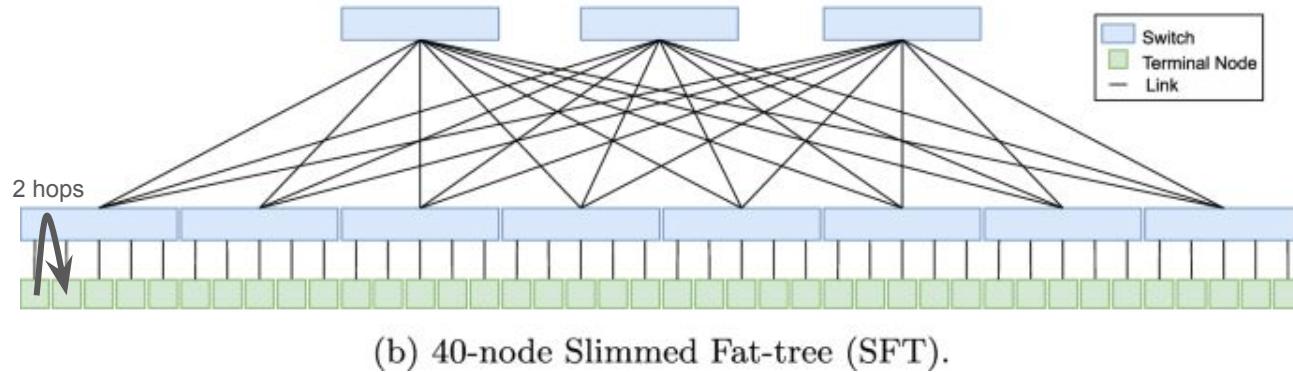
HYBRID_SHARD

- GPUs are organized into hierarchical structure



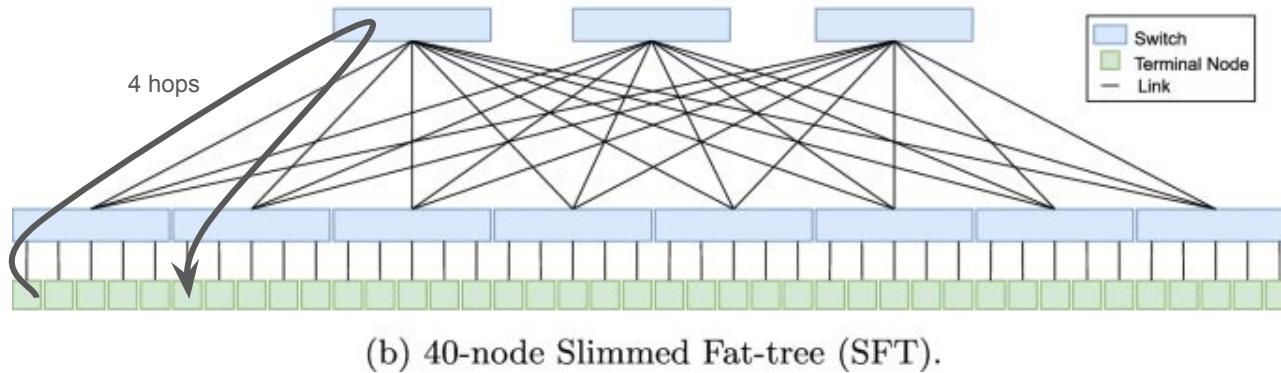
HYBRID_SHARD

- GPUs are organized into hierarchical structure



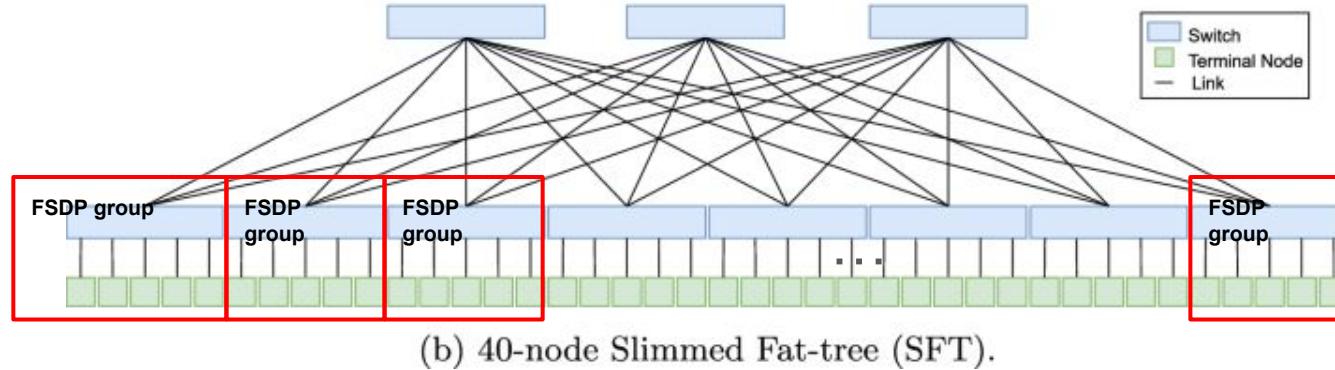
HYBRID_SHARD

- GPUs are organized into hierarchical structure



HYBRID_SHARD

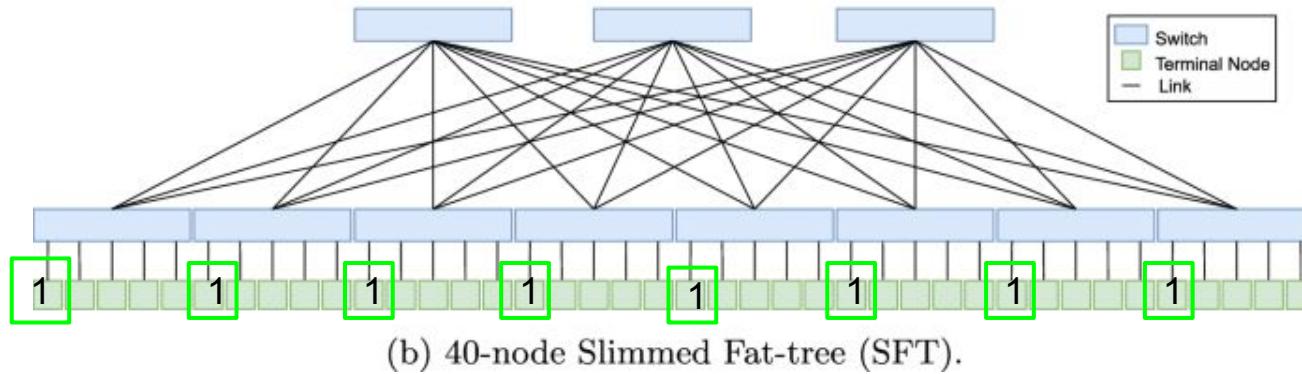
- All-Gather and Reduce-Scatter inside FSDP group



HYBRID_SHARD

- All-Reduce across cluster-wide DDP groups

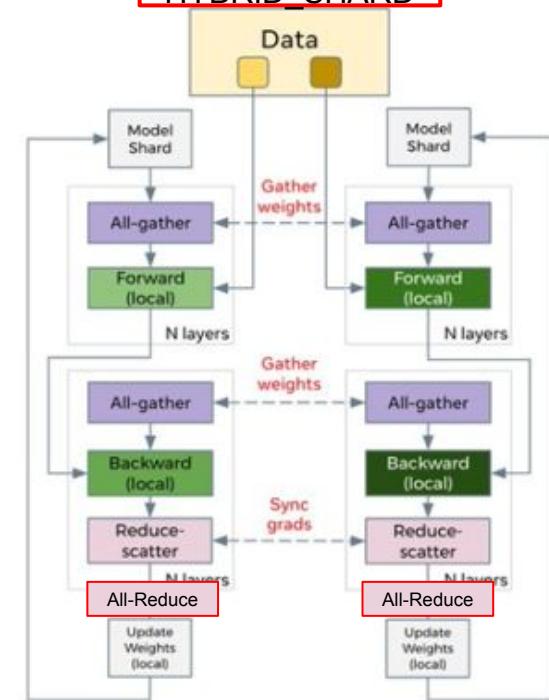
1 (All GPUs from this group own identical weights)



HYBRID_SHARD

- FSDP group size is usually 256-512 GPUs
- Number of DDP groups depends on size of cluster

Fully Sharded Data Parallel Training
HYBRID_SHARD



FSDP 2 Crash Course

RFC(Request for comments):

<https://github.com/pytorch/pytorch/issues/114299>

- Per-parameter sharding
- Torch.Compile compatibility

FSDP 2 Crash Course

- Per-parameter sharding
- In FSDP1 optimizer only saw part of FlatParam
- Optimizer couldn't treat different parameters in Module differently

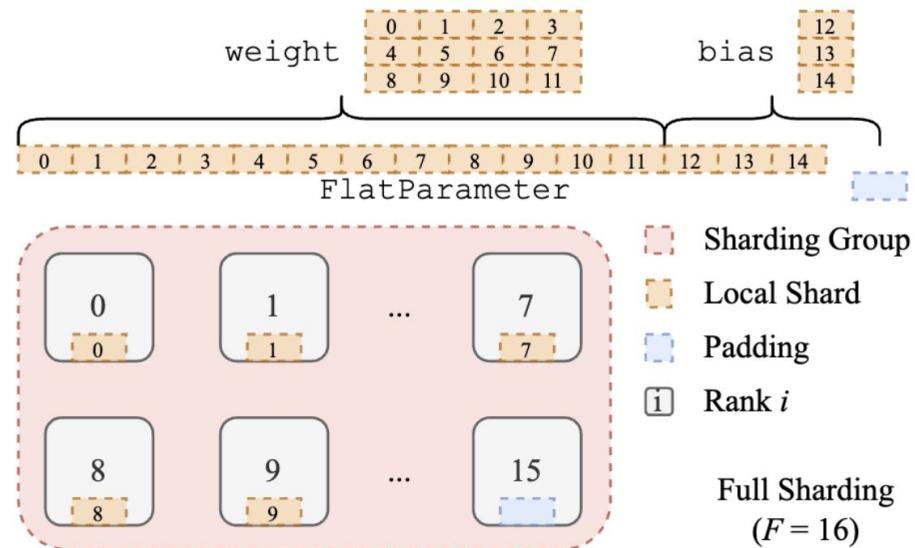


Figure 3: Full Sharding Across 16 GPUs

ФАЙЛ НЕЭФФЕКТИВНОСТИ

- Каждый DataParallel ранг владеет частью fp32 весов
- Перед forward и backward мы собираем bf16 версию весов
- Как только посчитался градиент для всех весов в модуле, запускаем Reduce-Scatter
- Каждый ранг получают градиенты для своей части весов и обновляет их

