FSDP семинар

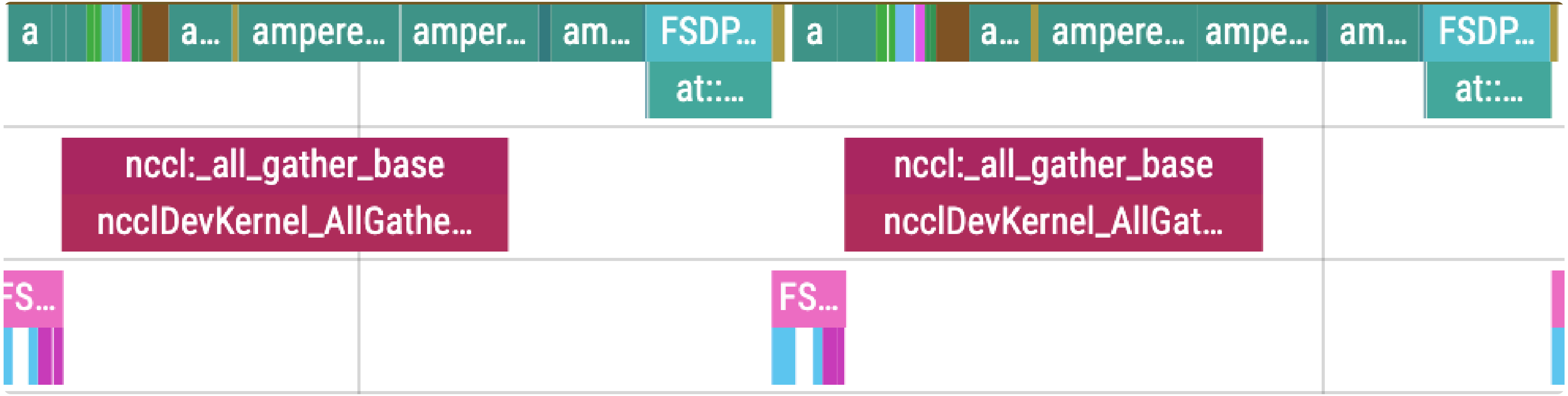
Plan

- Prerequisites: CUDA streams / events, DeviceMesh, DTensor
- FSDP2: interface, options, internals
- PyTorch DCP, efficient garbage collection

CUDA streams and events

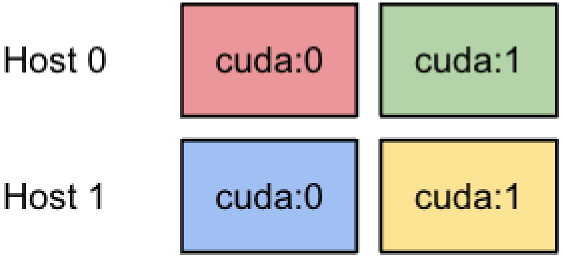
```python
all_gather_stream = torch.cuda.Stream()

 ...

# layer 3 unshard
with torch.cuda.stream(all_gather_stream):
    model.layers[3].all_gather()
    all_gather_event_3 = torch.cuda.Event()
    # or all_gather_stream.record_event()

# layer 2 forward
activations = model.layers[2](activations)

# layer 4 unshard
with torch.cuda.stream(all_gather_stream):
    model.layers[4].all_gather()
    all_gather_event_4 = torch.cuda.Event()

# layer 3 forward
torch.cuda.default_stream().wait_event(all_gather_event_3)
activations = model.layers[3](activations)

 ...
```

CUDA streams and events



DeviceMesh

DeviceMesh: The higher level abstraction that manages ProcessGroups



Host 0	cuda:0	cuda:1
Host 1	cuda:0	cuda:1

2 host with 2 GPUs each, represented
as a 2-D mesh [[0, 1], [2, 3]]

DeviceMesh

```python
from torch.distributed.device_mesh import init_device_mesh

mesh_1d = init_device_mesh("cuda", mesh_shape=(8,), mesh_dim_names=("dp",))
mesh_2d = init_device_mesh("cpu", mesh_shape=(2, 8), mesh_dim_names=("dp", "tp"))
mesh_3d = init_device_mesh(
    "cuda",
    mesh_shape=(2, 2, 8),
    mesh_dim_names=("pp", "dp", "tp"),
)

dp_group = mesh_2d.get_group("dp")
dist.all_gather( ... , group=dp_group)

mesh_2d.get_local_rank("tp")

mesh = init_device_mesh("cpu", mesh_shape=(16,), mesh_dim_names=("world",))
mesh_dp_cp = mesh._unflatten(0, mesh_sizes=(4, 4), mesh_dim_names("dp", "cp"))
mesh_host = mesh._unflatten(0, mesh_sizes=(2, 8), mesh_dim_names("inter", "intra"))
```

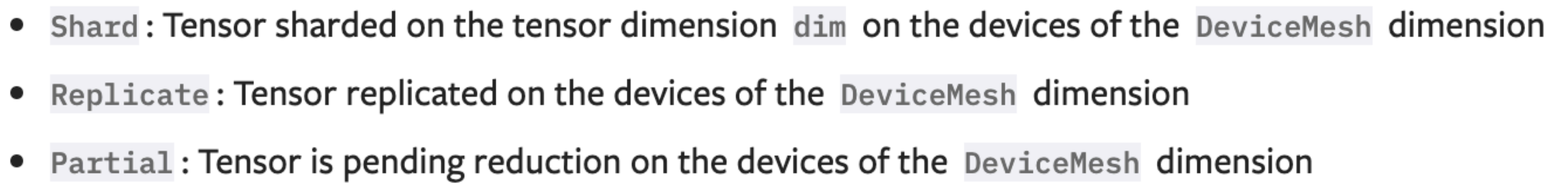
DTensor

```python
class DTensor:
    _local_tensor: torch.Tensor
    _spec:

@dataclass
class DTensorSpec:
    mesh: DeviceMesh
    placements: tuple[Placement, ...]  # (Shard(0), Replicate())

    # tensor meta will only be set during sharding propagation
    tensor_meta: TensorMeta | None = None  # (dtype, shape, stride)

    def redistribute(
        placements: Sequence[Placement]
    ) -> DTensor | torch.Tensor: ...

    def __torch_dispatch__(): ...
```

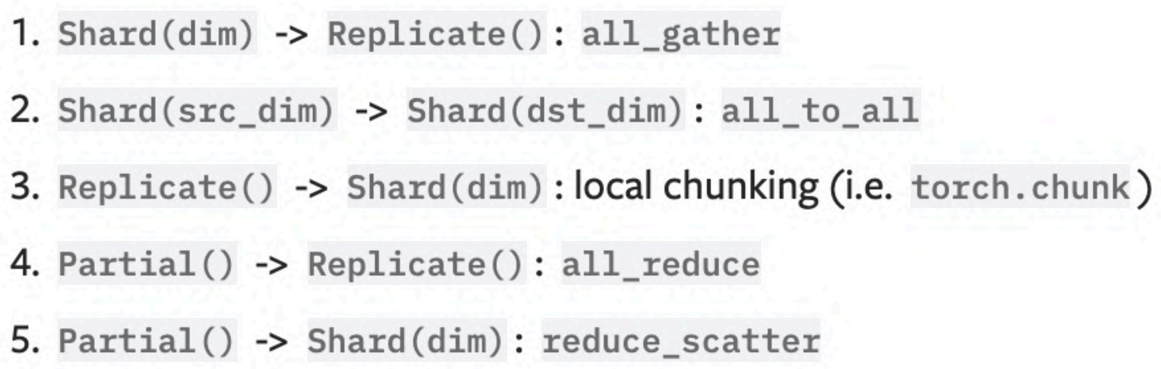
DTensor

```python
from torch.distributed.tensor import DTensor, distribute_tensor

mesh = init_device_mesh("cuda", mesh_shape=(8,), mesh_dim_names=("dp",))
big_tensor = torch.randn(1024, 4096)
placements = (Shard(dim=0),)

dtensor = distribute_tensor(
    big_tensor,
    device_mesh=mesh,
    placements=placements,
)
dtensor._local_tensor
dtensor.to_local() # .shape = (512, 4096)


shard = ... # .shape = (512, 4096)
dtensor = DTensor.from_local(
    shard,
    device_mesh=mesh,
    placements=placements,
) # .shape = (1024, 4096)

dtensor.redistribute(placements=(Replicate(),))
dtensor.full_tensor()
```

DTensor

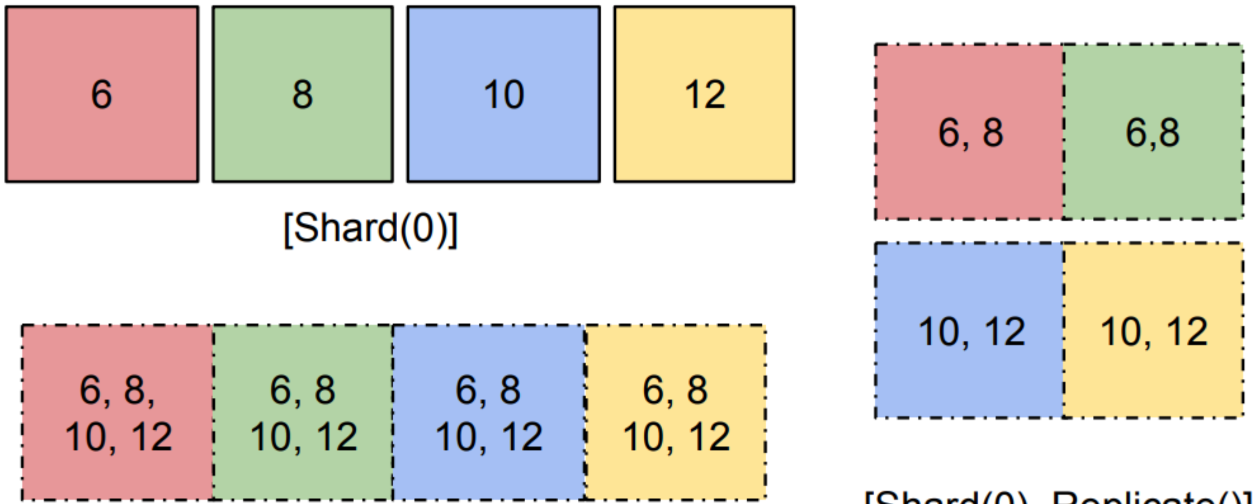


- `Shard` : Tensor sharded on the tensor dimension `dim` on the devices of the `DeviceMesh` dimension
- `Replicate` : Tensor replicated on the devices of the `DeviceMesh` dimension
- `Partial` : Tensor is pending reduction on the devices of the `DeviceMesh` dimension



1. `Shard(dim)` -> `Replicate()` : `all_gather`
2. `Shard(src_dim)` -> `Shard(dst_dim)` : `all_to_all`
3. `Replicate()` -> `Shard(dim)` : local chunking (i.e. `torch.chunk`)
4. `Partial()` -> `Replicate()` : `all_reduce`
5. `Partial()` -> `Shard(dim)` : `reduce_scatter`

DTensor



Data: [6, 8, 10, 12]

6

8

10

12

[Shard(0)]

6, 8,
10, 12

6, 8
10, 12

6, 8
10, 12

6, 8
10, 12

[Replicate()]

DeviceMesh: ([0, 1, 2, 3])

6, 8

6,8

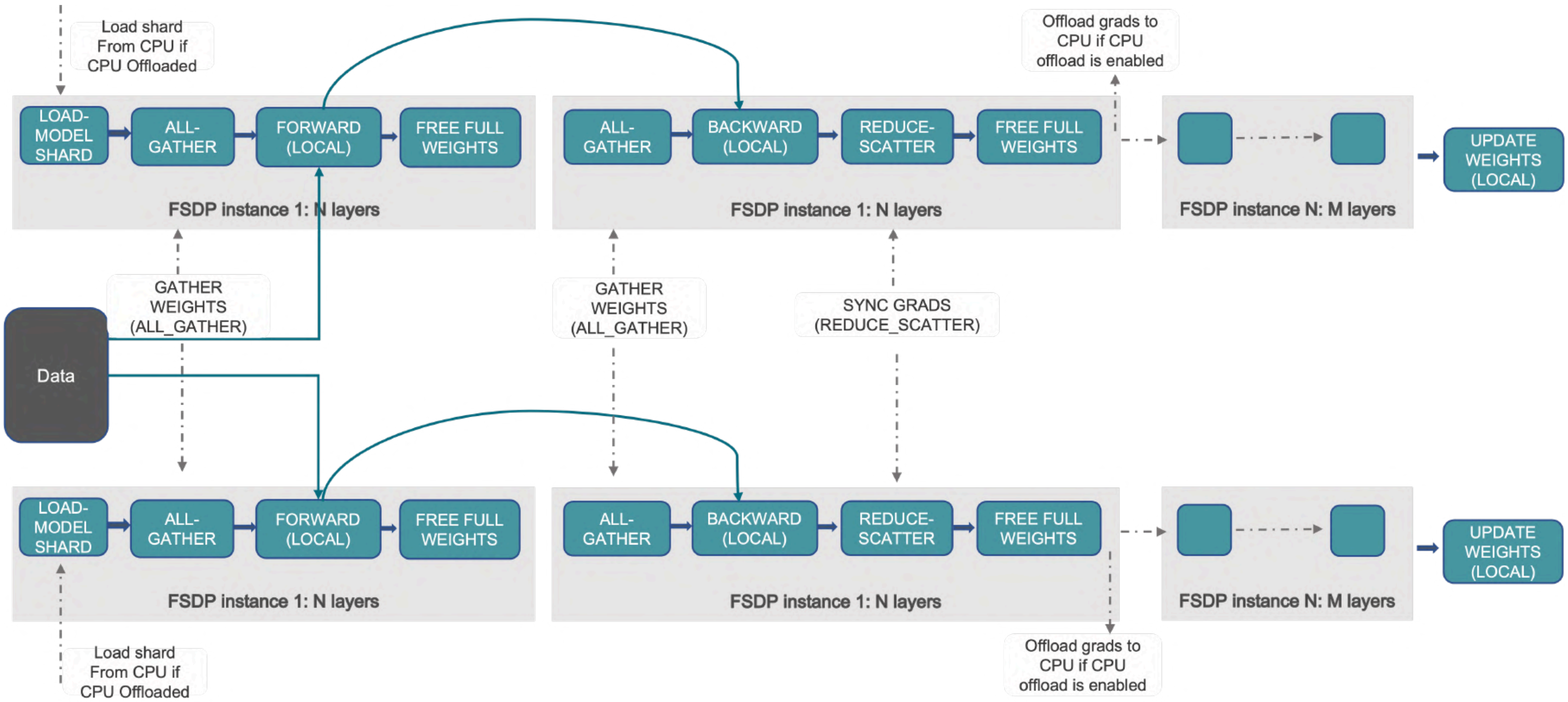
10, 12

10, 12

[Shard(0), Replicate()]

DeviceMesh: ([[0, 1], [2, 3]])

FSDP2



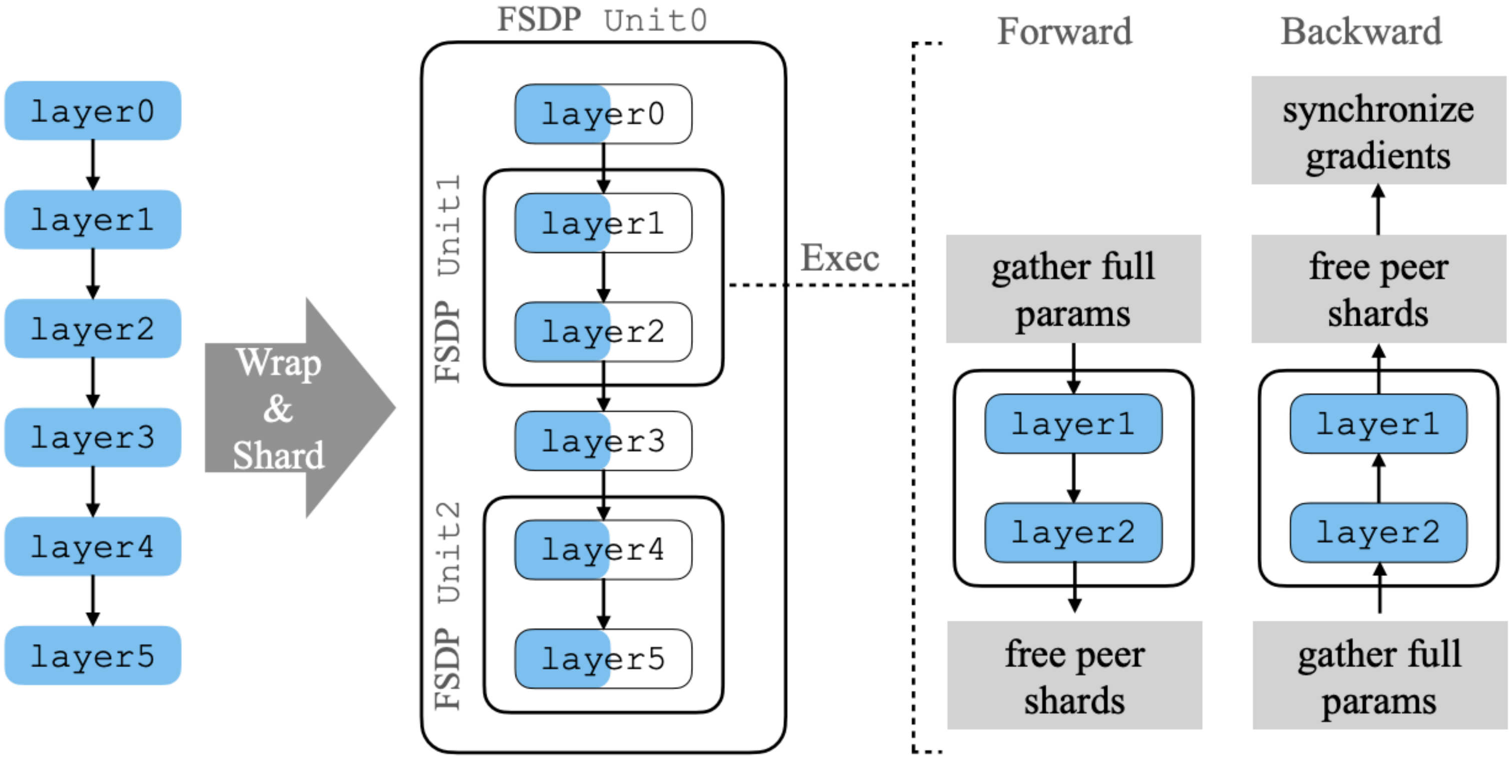
Load shard From CPU if CPU Offloaded
LOAD-MODEL SHARD
ALL-GATHER
FORWARD (LOCAL)
FREE FULL WEIGHTS
FSDP instance 1: N layers
ALL-GATHER
BACKWARD (LOCAL)
REDUCE-SCATTER
FREE FULL WEIGHTS
FSDP instance 1: N layers
Offload grads to CPU if CPU offload is enabled
FSDP instance N: M layers
UPDATE WEIGHTS (LOCAL)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

FSDP2

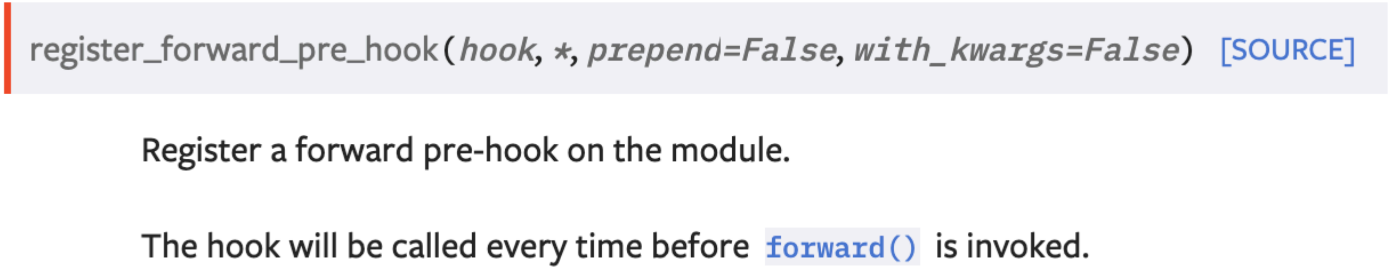
```python
from torch.distributed.fsdp import fully_shard

mesh_2d = init_device_mesh(
    "cuda",
    mesh_shape=(2, 8),
    mesh_dim_names=("dp", "tp"),
)
model = Model()

for layer in model.layers:
    fully_shard(
        module, # (module1, module2)
        mesh=dp_mesh,
        reshard_after_forward=True, # ZeRO-3
        mp_policy=MixedPrecisionPolicy(
            param_dtype=torch.float16,
            reduce_dtype=torch.float32,
        ),
        offload_policy=CPUOffloadPolicy(),
    )

fully_shard(model, ... )
```

```python
for step in ... :
    for gas_step in ... :
        is_last_backward = gas_step == num_gas_steps - 1
        # ZeRO-2
        model.set_reshard_after_backward(is_last_backward)
        # ZeRO-1
        model.requires_gradient_sync(is_last_backward)

    loss = loss_fn(model(inputs), targets)
    ...
```

FSDP2


layer0
layer1
layer2
layer3
layer4
layer5
Wrap & Shard
FSDP Unit0
layer0
FSDP Unit1
layer1
layer2
layer3
FSDP Unit2
layer4
layer5
Exec
Forward
gather full params
layer1
layer2
free peer shards
Backward
synchronize gradients
free peer shards
layer1
layer2
gather full params

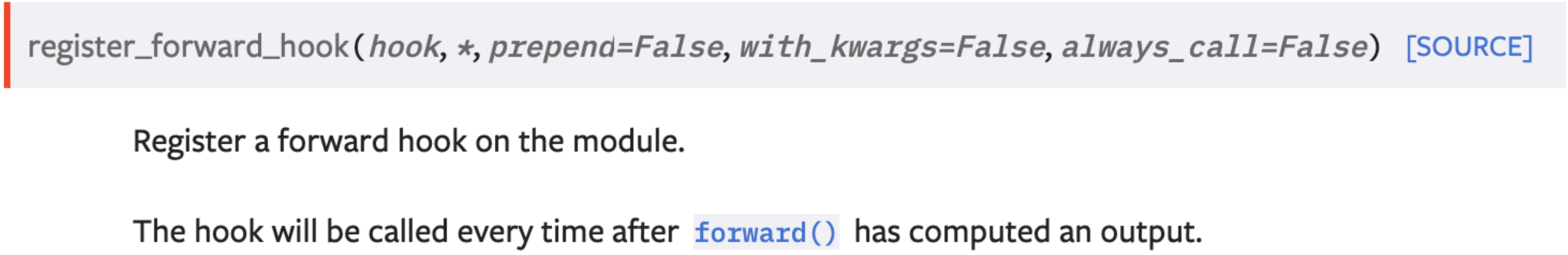
FSDP2 — hooks



register_forward_pre_hook(*hook*, ***, *prepend=False*, *with_kwargs=False*) [SOURCE]

Register a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked.



register_forward_hook(*hook*, ***, *prepend=False*, *with_kwargs=False*, *always_call=False*) [SOURCE]

Register a forward hook on the module.

The hook will be called every time after `forward()` has computed an output.

FSDP2 — pre-forward

```python
def pre_forward(module, args):
    module.unshard() # in all-gather stream
    module.wait_for_unshard() # sync compute (default) stream with all-gather stream
    module._register_post_backward_hook(args)
    return args

def unshard(module):
    with torch.cuda.stream(all_gather_stream):
        module.all_gather_result = module.all_gather()
    module.all_gather_event = all_gather_stream.record_event()

def wait_for_unshard(module):
    torch.cuda.default_stream().wait_event(module.all_gather_event)
    module.set_unsharded_params(module.all_gather_result)

def fully_shard(module, ...):
    ...
    module.register_forward_pre_hook(pre_forward)
```

FSDP2 — post-forward

```python
def post_forward(module, args, output):
    module.reshard()
    module._record_post_forward()
    module._register_pre_backward_hook(output)
    return output

def reshard(module):
    module.set_sharded_params() # and free unsharded params

def _record_post_forward(module):
    post_forward_index = len(module.comm_ctx.post_forward_order)
    module.comm_ctx.post_forward_order.append(module)
    module._post_forward_indices.append(post_forward_index)

def fully_shard(module, ... ):
    ...
    module.register_forward_hook(post_forward)
```

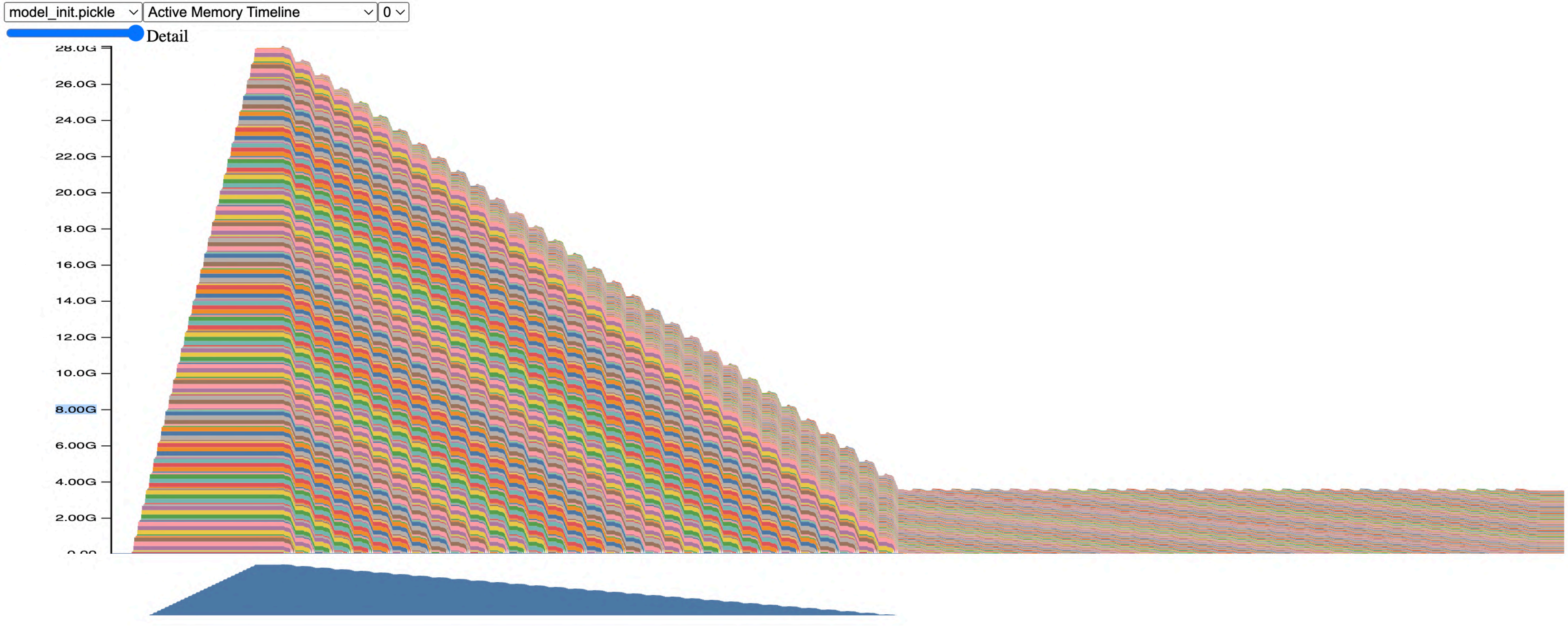
FSDP2 — pre-backward

```python
def pre_backward(module, *unused):
    module.unshard()  # no-op if prefetched
    module.wait_for_unshard()
    module._backward_prefetch()

def _backward_prefetch(module):
    curr_index = module._post_forward_indices.pop()
    target_index = curr_index - 1
    target_module = self.comm_ctx.post_forward_order[target_index]
    target_module.unshard()

def _register_pre_backward_hook(self, output):
    for t in output:
        if torch.is_tensor(t) and t.requires_grad:
            t.register_hook(self._pre_backward)
    return output
```

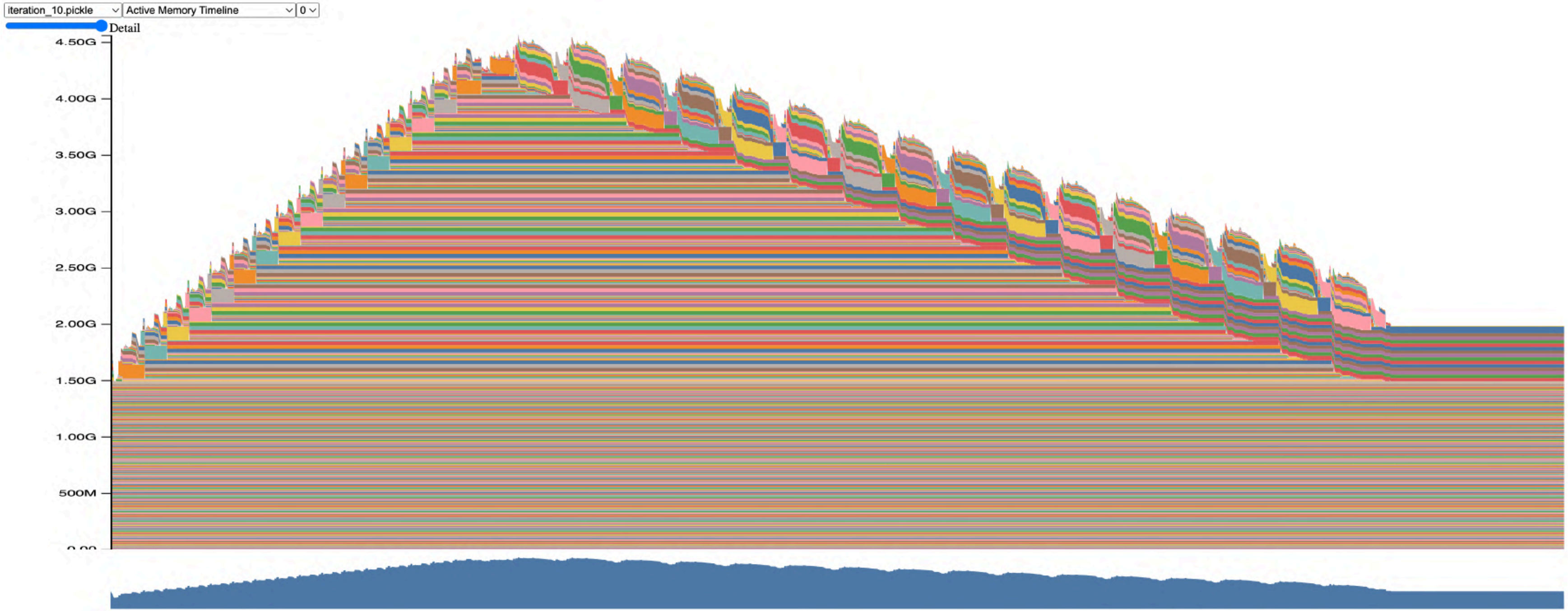
FSDP2 — post-backward

```python
def post_backward(module, *unused: Any):
    if module.reshard_after_backward:
        module.reshard()
    if module.reduce_grads:
        reduce_scatter_stream.wait_stream(torch.cuda.default_stream())
        with torch.cuda.stream(reduce_scatter_stream):
            module.reduce_scatter_grads()
        reduce_event = reduce_scatter_stream.record_event()

def _register_post_backward_hook(module, args):
    RegisterPostBackwardFunction.apply(self, *args)

class RegisterPostBackwardFunction(torch.autograd.Function):
    @staticmethod
    def forward(ctx, module, *inputs):
        ctx.module = module
        return inputs

    @staticmethod
    def backward(ctx, *grads):
        module.post_backward()
        return (None,) + grads
```

FSDP2 — memory



FSDP2 — memory

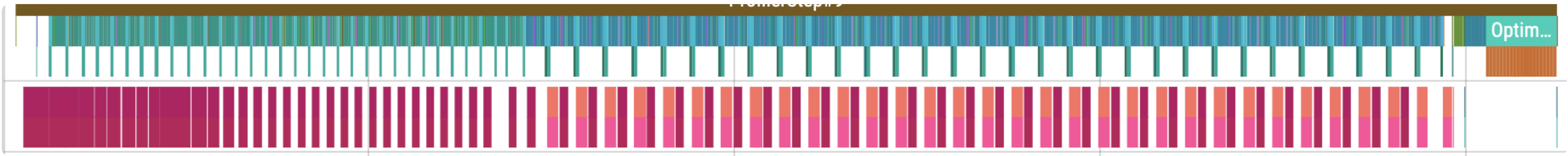


Computation / communication overlap

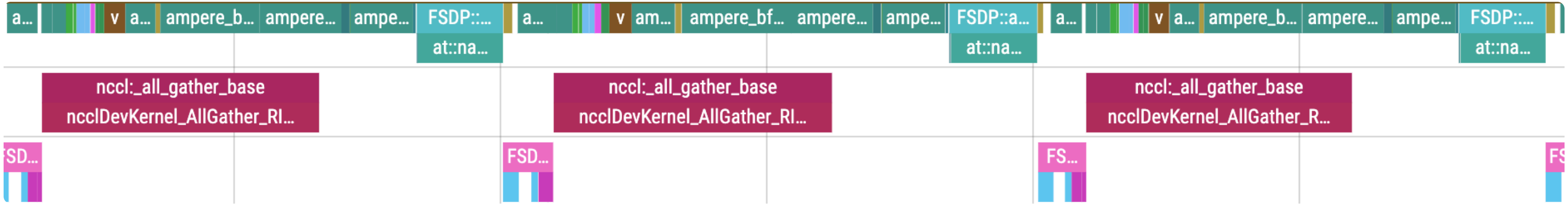
- Implicit prefetching
  - в `pre_forward`
- Explicit prefetching
  - в `pre_backward`
  - можно задать руками

```
module.set_modules_to_forward_prefetch(modules)
module.set_modules_to_backward_prefetch(modules)
```

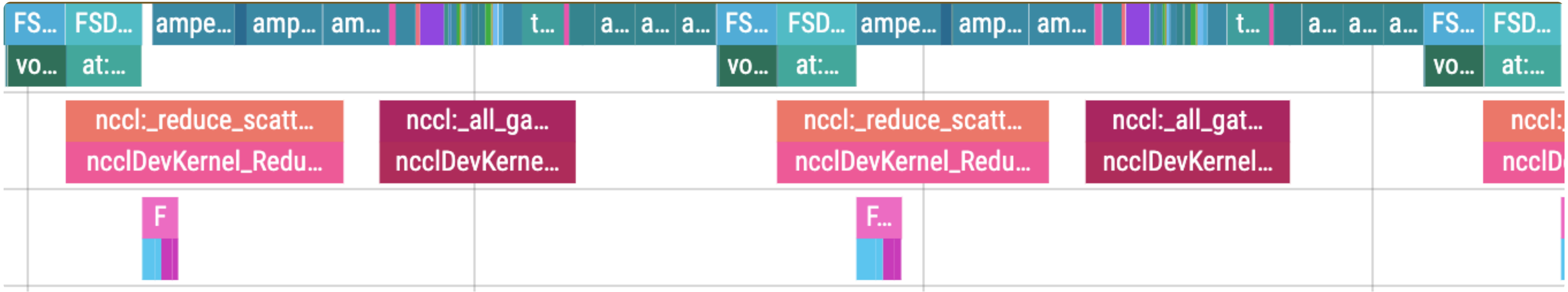
Подробнее про работу со стримами



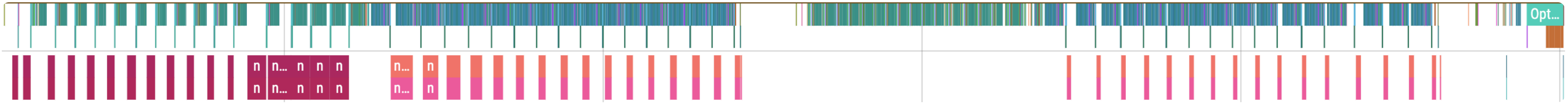
Подробнее про работу со стримами — forward



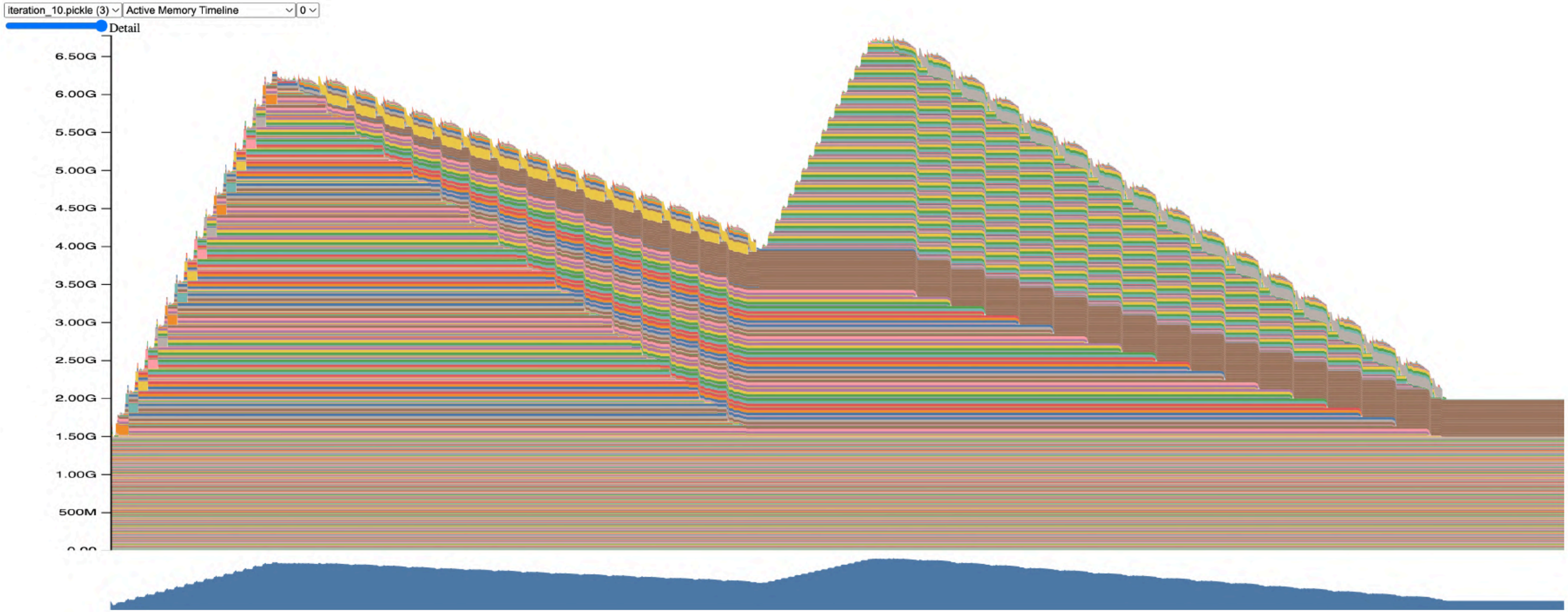
Подробнее про работу со стримами — backward



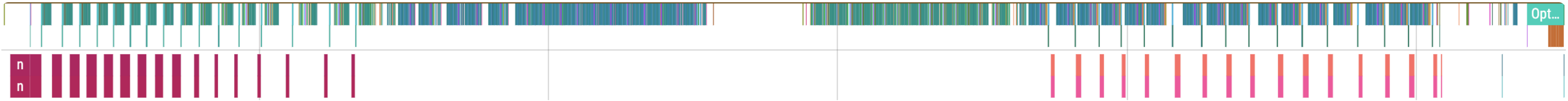
ZeRO-2



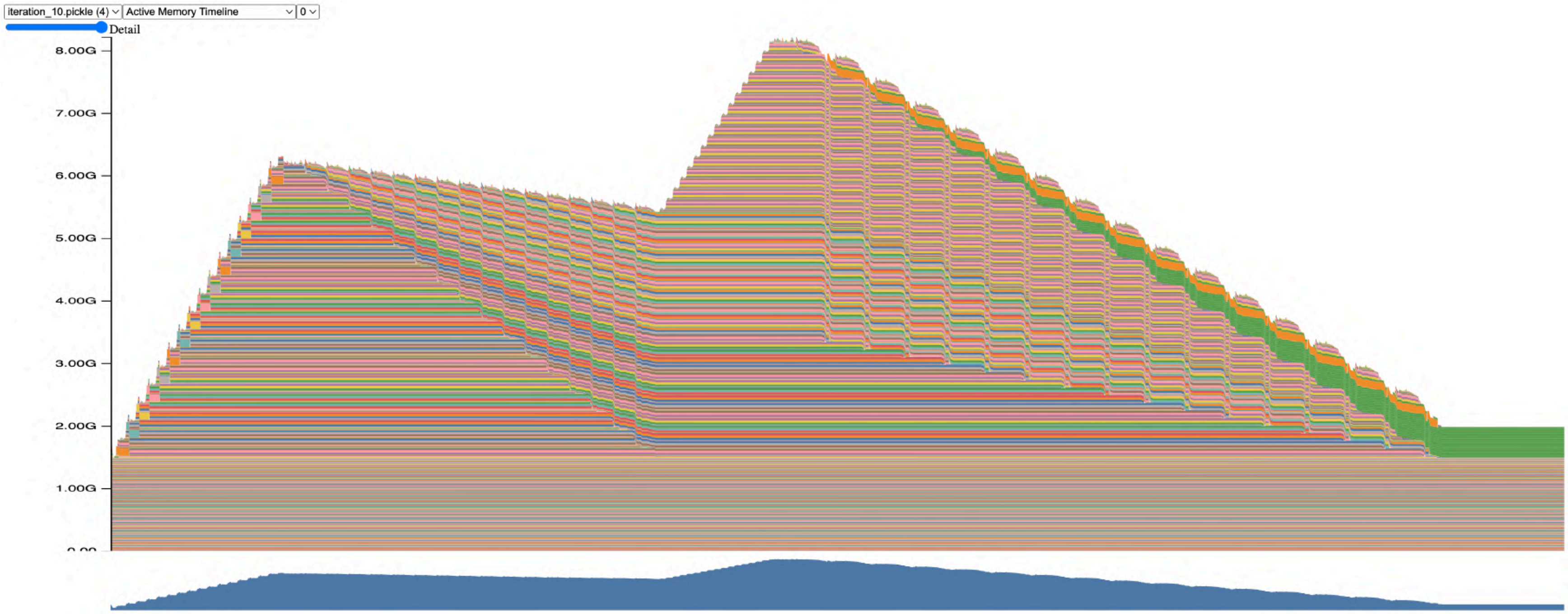
ZeRO-2


iteration_10.pickle (3)
Active Memory Timeline
0
Detail
6.50G
6.00G
5.50G
5.00G
4.50G
4.00G
3.50G
3.00G
2.50G
2.00G
1.50G
1.00G
500M
0.00

ZeRO-1



ZeRO-1


iteration_10.pickle (4)
Active Memory Timeline
0
Detail
8.00G
7.00G
6.00G
5.00G
4.00G
3.00G
2.00G
1.00G
0.00

HSDP

```
mesh_2d = init_device_mesh(
    "cpu",
    mesh_shape=(2, 8),
    mesh_dim_names=("dp_replicate", "dp_shard"),
)

fully_shard(
    module,
    mesh=mesh_2d,
    ...
)
```

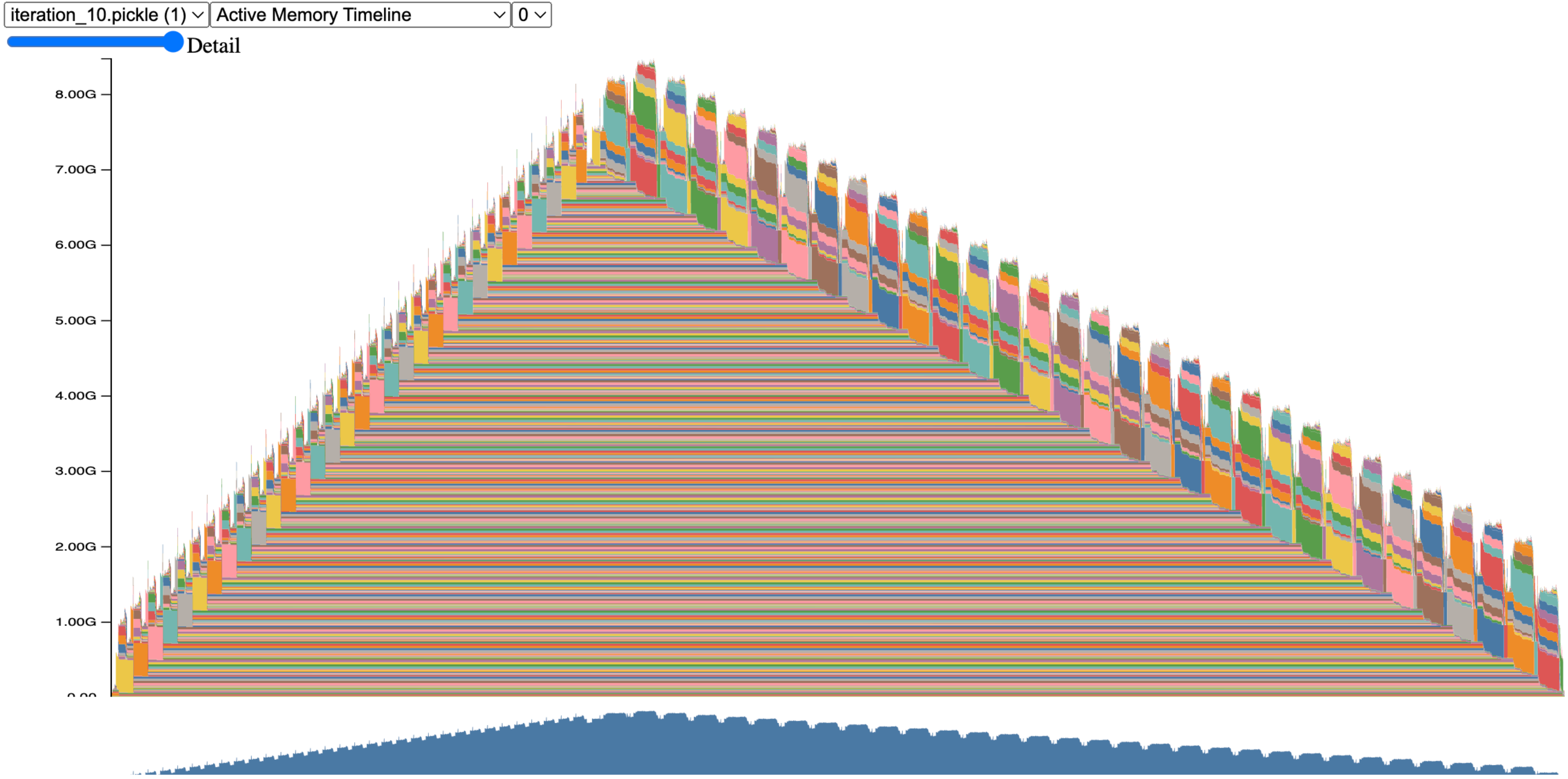
- логика становится заметно сложнее, показывать не буду(

CPU offloading

- ZeRO-Offload

```python
with torch.device("cpu"):
    model = Model()

fully_shard(
    module,
    ...
    offload_policy=CPUOffloadPolicy(),
)

def unshard(module):
    sharded_param = sharded_param.to(
        device,
        non_blocking=True,
    )
    ...
    module.all_gather()

def post_backward(module):
    new_sharded_grad = new_sharded_grad.to(
        torch.device("cpu"),
        non_blocking=True
    )
```

CPU offloading

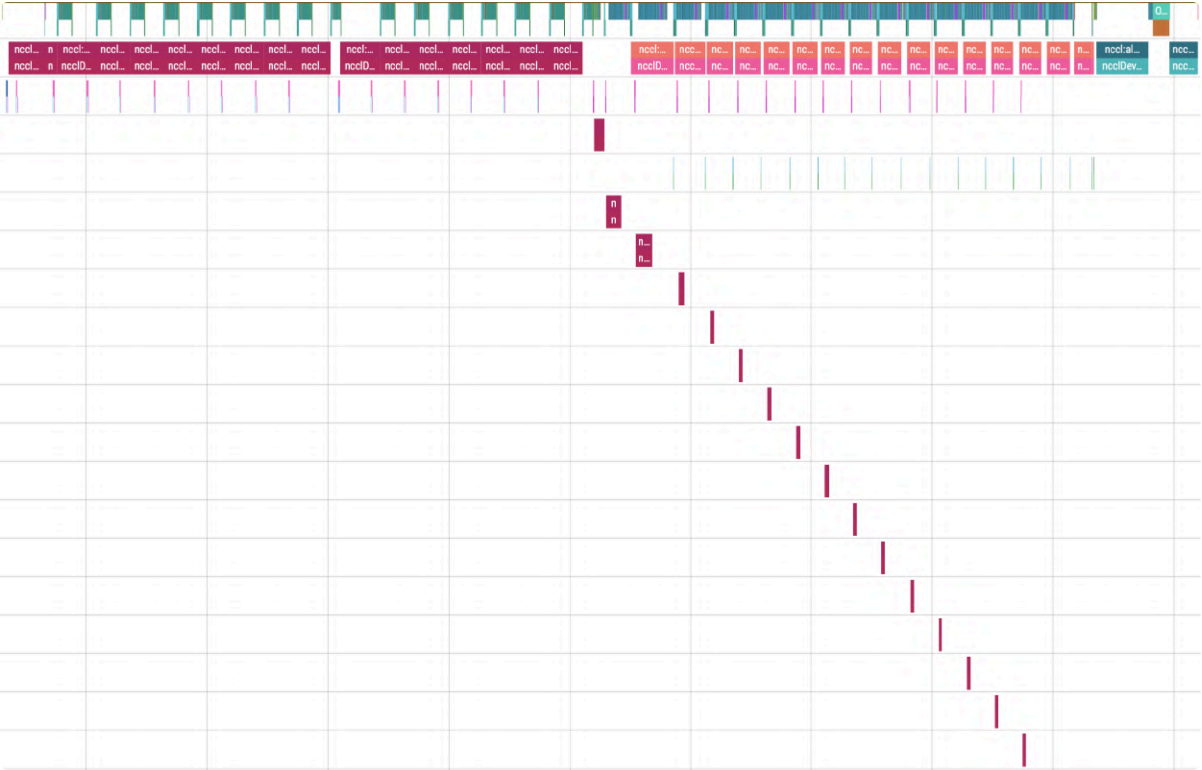


hpZ

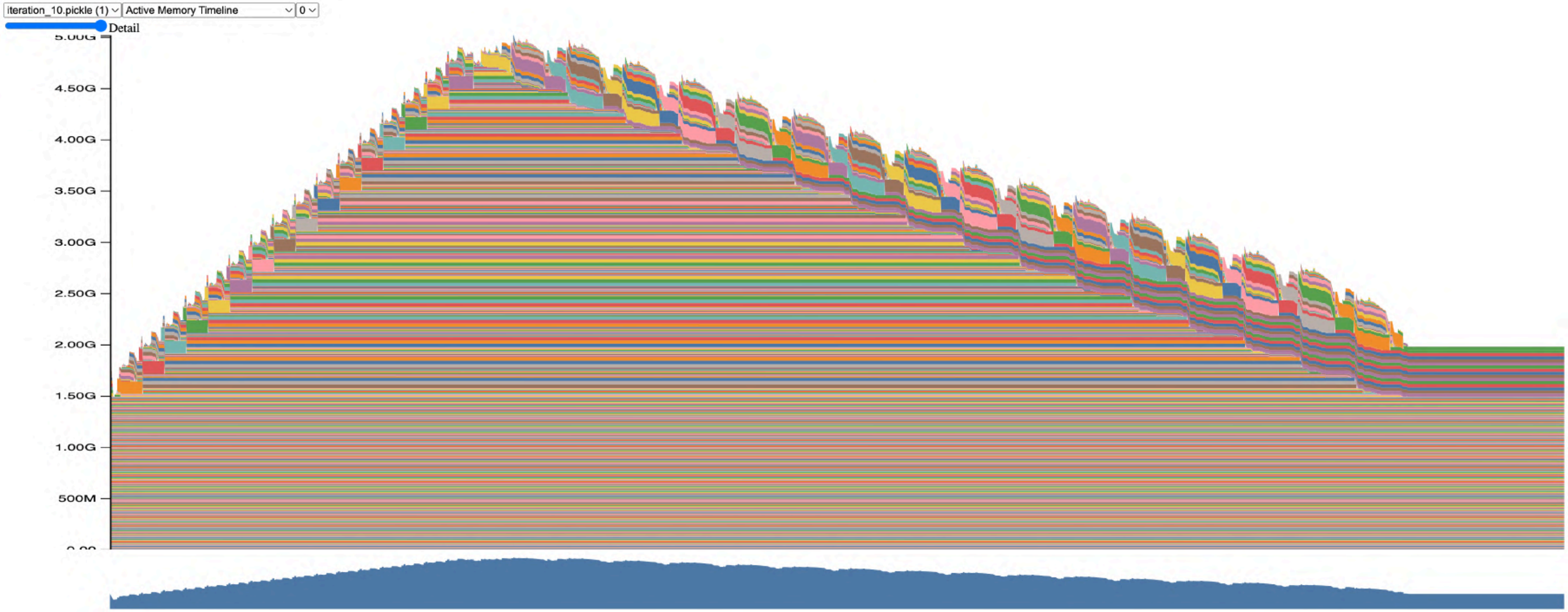
- ZeRO++

```
mesh = init_device_mesh(
    "cuda",
    mesh_shape=(16,),
    mesh_dim_names=("dp",),
)
fully_shard(
    module,
    mesh,
    ...
    reshard_after_forward=8,
)
```

hpZ



hpZ


iteration_10.pickle (1)
Active Memory Timeline
0
Detail
5.00G
4.50G
4.00G
3.50G
3.00G
2.50G
2.00G
1.50G
1.00G
500M
0.00

PyTorch DCP

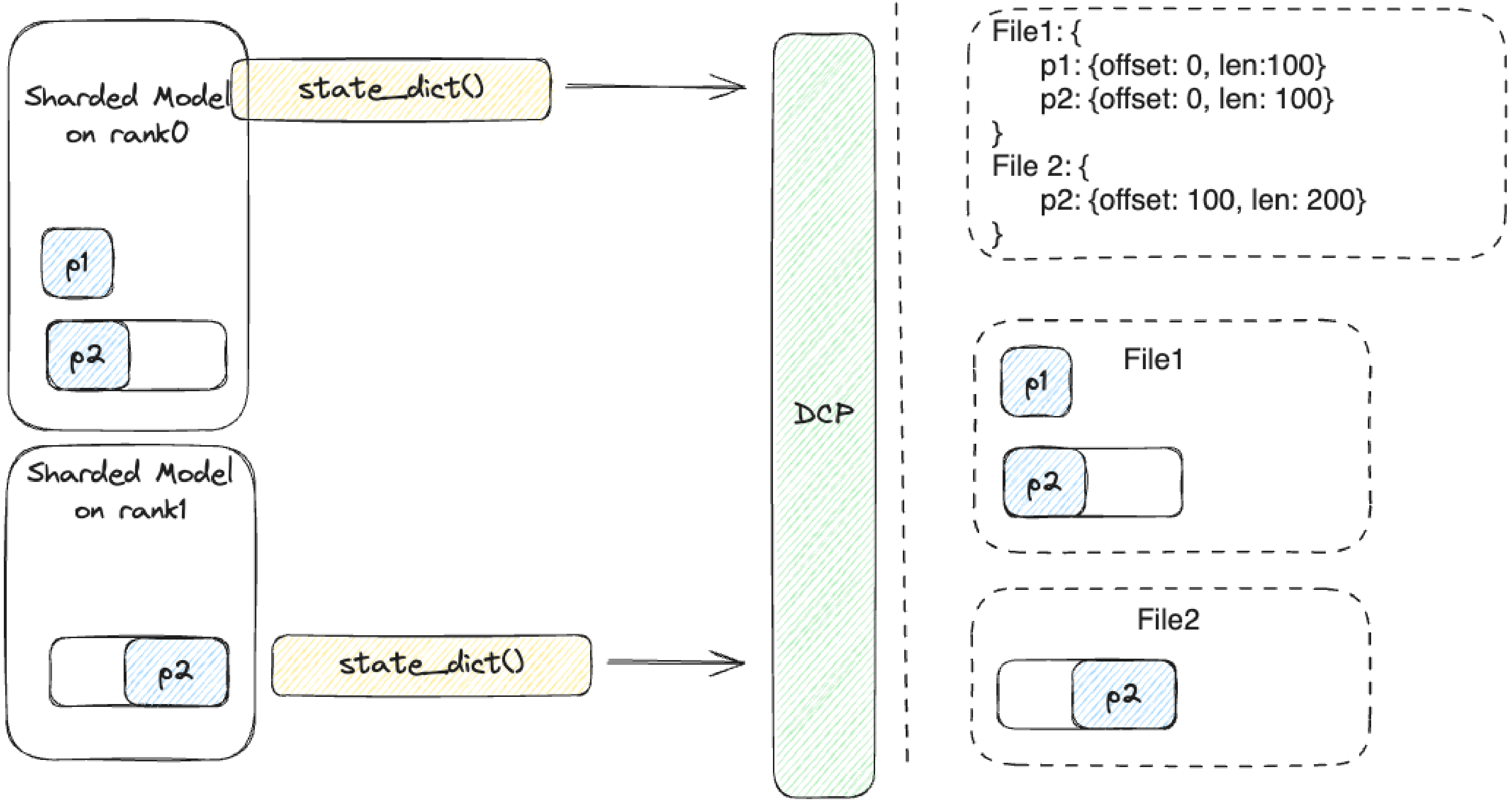
- два вида `state_dict`
  - `SHARDED_STATE_DICT`
  - `FULL_STATE_DICT`
- в FSDP2 всегда sharded, но состоит из DTensor-ов
  - с помощью `.redistribute()` можно менять шардирование чекпоинта
- DCP умеет эффективно отгружать чекпоинты с минимальным оверхедом

PyTorch DCP

```python
import torch.distributed.checkpoint as dcp
model = Model()
fully_shard(model)
optimizer = Optimizer(model.parameters())

state_dict = {
    "model": model.state_dict(),
    "optimizer": optimizer.state_dict()
}
dcp.state_dict_saver.save(state_dict)
dcp.state_dict_loader.load(state_dict)
```

- truthfully it’s a bit more complicated

PyTorch DCP

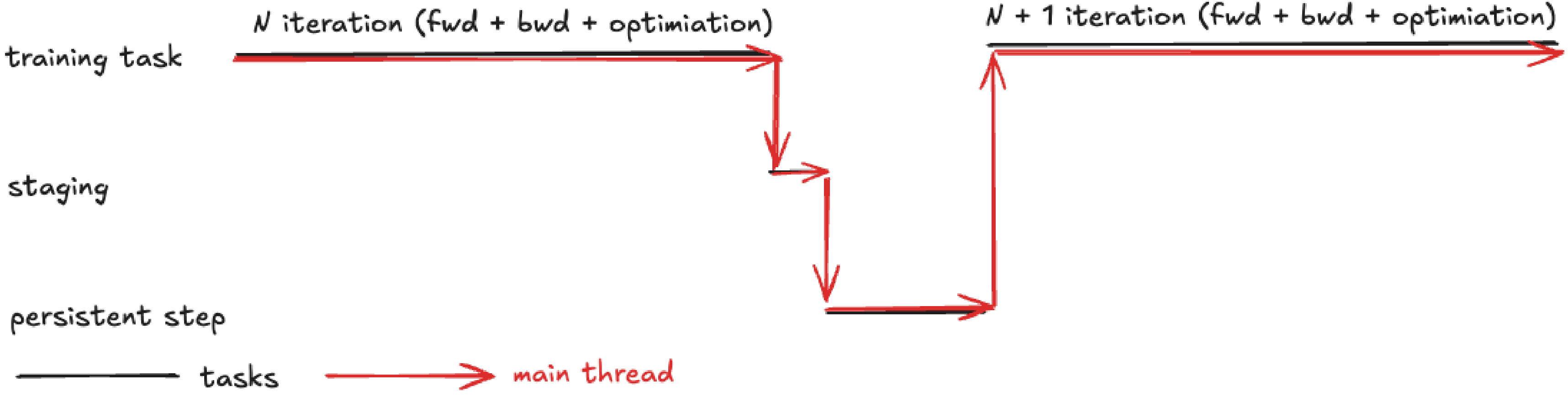


```
File1: {
    p1: {offset: 0, len:100}
    p2: {offset: 0, len: 100}
}
File 2: {
    p2: {offset: 100, len: 200}
}
```

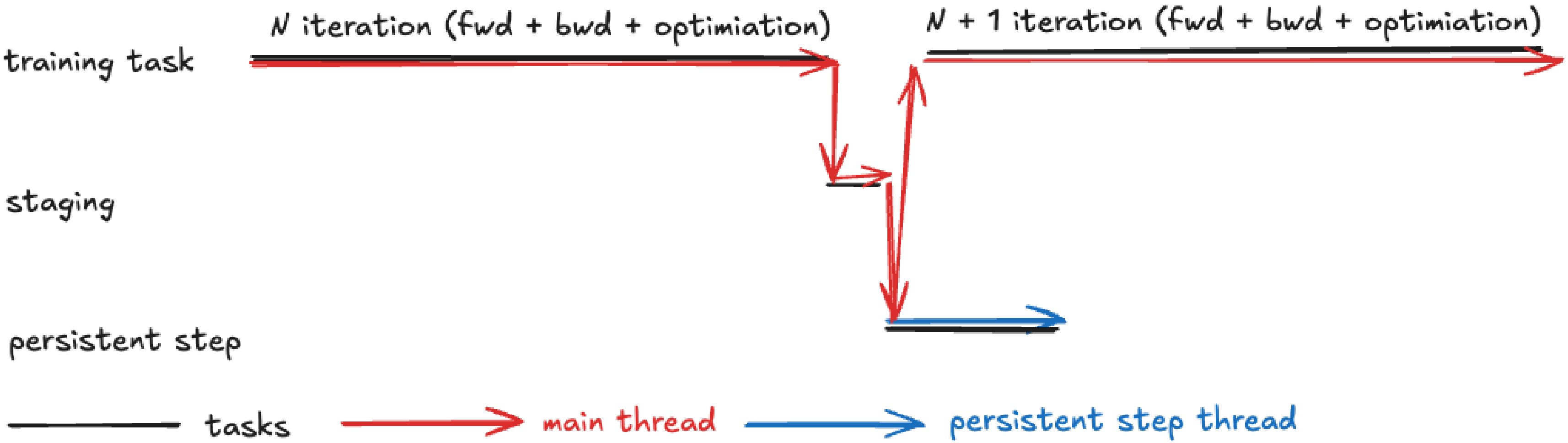
File1

File2

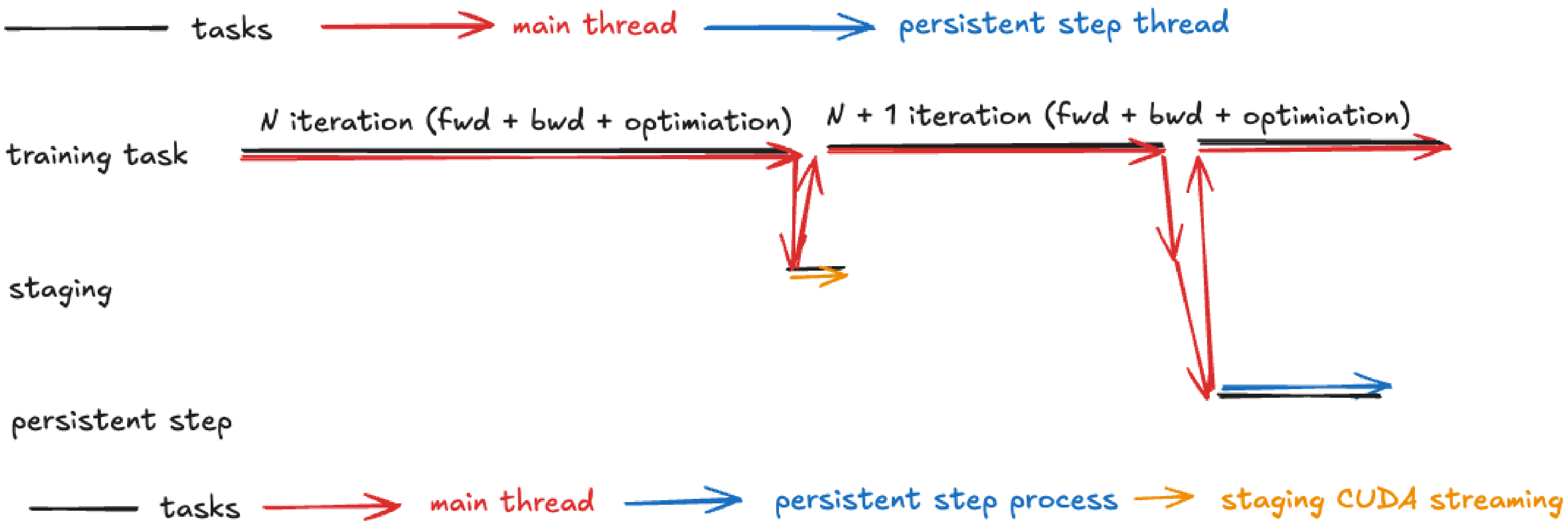
PyTorch DCP


N iteration (fwd + bwd + optimiation)
N + 1 iteration (fwd + bwd + optimiation)
training task
staging
persistent step
tasks
main thread

PyTorch DCP


N iteration (fwd + bwd + optimiation)
N + 1 iteration (fwd + bwd + optimiation)
training task
staging
persistent step
tasks
main thread
persistent step thread

PyTorch DCP


tasks
main thread
persistent step thread
N iteration (fwd + bwd + optimiation)
N + 1 iteration (fwd + bwd + optimiation)
training task
staging
persistent step
tasks
main thread
persistent step process
staging CUDA streaming

Garbage collection tuning

```
gc.disable()
gc.collect(1)

... init

for step in ...:
    if step > 1 and step % _gc_freq == 0:
        gc.collect(1)

    ... step
```

Extras

- SimpleFSDP
- `unshard_in_backward`
- meta device init
- compile