



CONFERENCE

—NORTH AMERICA 2026—

Future of Distributed Communication in PyTorch

New APIs for Fault Tolerance, RDMA and Extensibility

Tristan Rice, Kapil Sharma
Meta, PyTorch Distributed

Distributed is changing

- Key influences
 - Heterogeneous hardware: NVLink, InfiniBand, EFA, NVMe-oF
 - RL and inference need one-sided, zero-copy transfers
 - Expanding number of hardware and backends
 - "power users" – agents are able to optimize
- Refined focus
 - Improving existing APIs
 - Cleaning up tech debt
 - Focusing on flexibility, performance and control
 - Enabling new features, hardware and backends
 - All primitives needed for projects like torchtitan+TitanRL out of the box

Key new APIs

- Windows & Transports: one-sided RDMA put/get
- Collective Hooks: flight recorder, NaN checks on every backend
- Backend Extensibility: pip-installable and Python backends
- Clean Interfaces: `_single` naming, functional backward, `weights_only`
- Rewritten NCCL Backend: the new default for "nccl"
- Collective Configs: per-collective NCCL options
- Fault Tolerance: live reconfigure after rank failures

Incubated in torchcomms, upstream in PyTorch

- Features proven in torchcomms at scale
- Now upstreamed into `torch.distributed`
- Zero migration required
- No extra dependency for these features





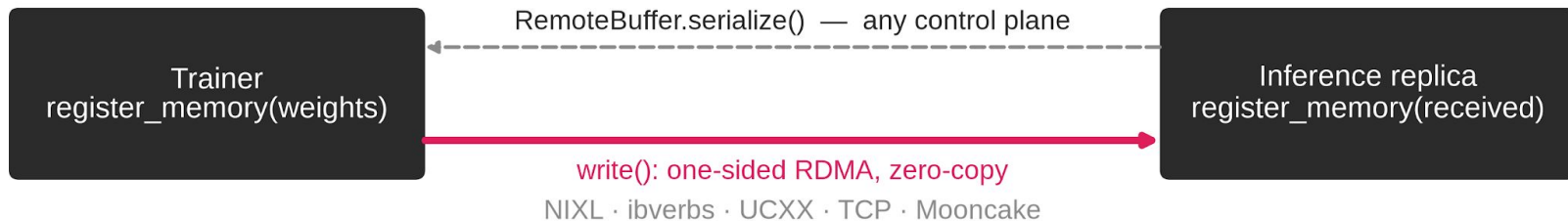
Into the weeds

RDMA

- One-sided: write remote memory, no matching receive
- Transports `dist._transport`: any peer, no process group
- Windows `dist._new_window`: ranks in a group, backend-native

torch.distributed._transport (new, experimental)

- point-to-point, no process group
- `bind()` / `connect(peer_url)`, `new_transport_rank(peer_rank)`
- `write / read, sync, async (async_op=True)` or `asyncio`
- Backends plug in via entry point `torch.distributed.transports`
- Provides a standard interface for torch* projects to access transports such as NIXL (torchstore, torchft, torchtitan, pipelining, etc)



Transport example: RL weight sync

```
from torch.distributed._transport import new_transport, RemoteBuffer

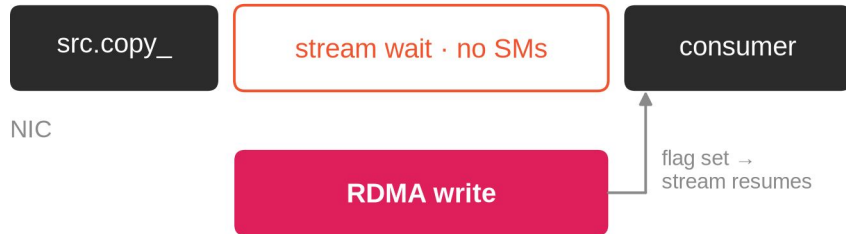
with new_transport("nixl") as trainer, new_transport("nixl") as replica:
    t_url, r_url = trainer.bind(), replica.bind()
    trainer.connect(r_url)
    replica.connect(t_url)
    src = trainer.register_memory(weights)
    dst = replica.register_memory(received)
    blob = dst.to_remote_buffer().serialize() # any control plane
    remote = RemoteBuffer.deserialize(blob)
    trainer.write(src.to_view(), remote, async_op=True).wait()
```

Transfers on CUDA streams and graphs

- write_stream / read_stream: ordered on a CUDA stream
- Return once enqueued; no kernel holds SMs
- Capture with torch.cuda.graph; each replay sends a new transfer
- Works for any host-driven backend
- Native stream backends can override

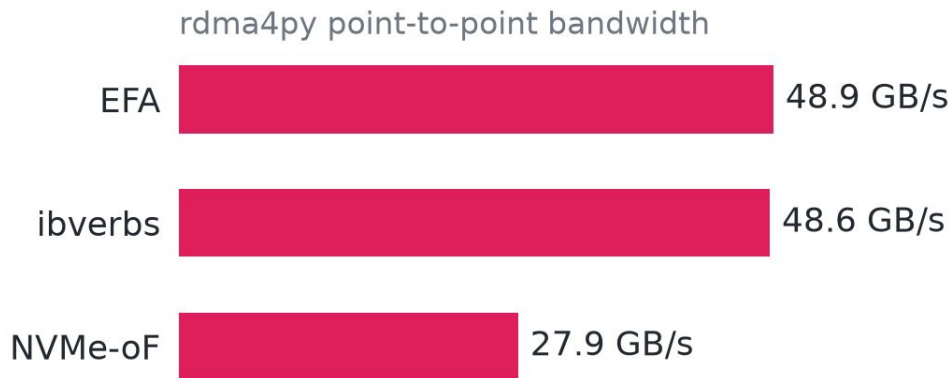
```
graph = torch.cuda.CUDAGraph()
with torch.cuda.graph(graph):
    src.copy_(weights)
    t.write_stream(src_view, remote)
for step in range(steps):
    train_step()
    graph.replay()  # new transfer
```

CUDA stream



Transport backends

- Landed: core API, NIXL, Store bootstrap, CUDA streams
- Prototypes: TCP, UCXX, TorchComms, rdma4py (ibverbs/EFA), Mooncake
- Common benchmark harness across backends



Window APIs

- One-sided put + signal between ranks
- `dist._supports_window(group)` — capability check
- `win = dist._new_window(tensor, group)` — collective registration
- Zero-copy over NVLink, InfiniBand, EFA
- Currently supports ncclPut / signal

```
be = dist.get_backend_impl(device=dev)
pool = torch.cuda.MemPool(be.mem_allocator)
with torch.cuda.use_mem_pool(pool):
    buf = torch.zeros(n * world, device=dev)
win = dist._new_window(buf) # collective
dst = (rank + 1) % world
src = (rank - 1) % world
inp = torch.full([n], rank, device=dev)
win.put(inp, dst, dst * n, True).wait()
win.wait_signal(src, True).wait()
```



Extensibility & API Improvements

Collective Hooks

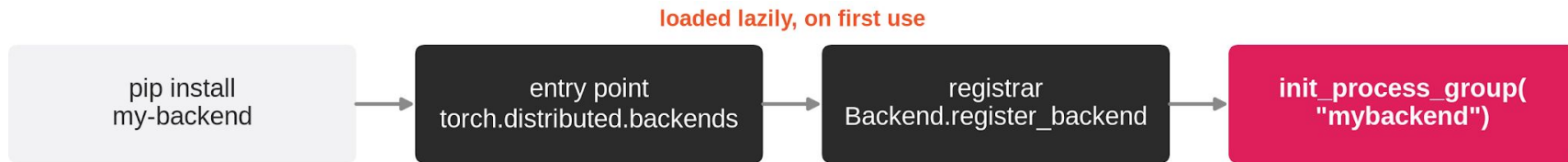
- Added hooks for observability, makes it easy to write custom integrations
- Built-in: Flight Recorder, NaN check, health check
- Works across all backends, not just NCCL

```
from torch._C._distributed_c10d import (  
    FlightRecorderHook, NanCheckHook)  
fr = FlightRecorderHook.attach(pg)  
nan = NanCheckHook.attach(pg)  
# custom hook  
def on_start(a):  
    log.info("%s %d", a.name, a.op_id)  
pg.register_pre_hook(1, on_start)
```

Backends as pip packages

- All backends, builtins too, register via `torch.distributed.backends`
- Loaded on first use — no patches, no forks
- Transports use the same model: `torch.distributed.transports`

```
[project.entry-points."torch.distributed.backends"]  
mybackend = "my_pkg:register"
```



Builtins take the same path: gloo · nccl · nccl-lazy · nccl-legacy · mpi · ucc · xccl

Python Backends

- PyBackend replaces PyProcessGroup
- Supports all collectives and operations
- Python makes it much easier to customize backend behavior
- Symmem + nccl4py prototypes

<https://github.com/pytorch/pytorch/blob/main/torch/distributed/nccl4py/backend.py>

https://docs.pytorch.org/docs/main/distributed/custom_backends.html

```
import torch
import torch.distributed as dist

class MyWork(dist.Work):
    def __init__(self, result):
        super().__init__()
        self._future = torch.futures.Future()
        self._future.set_result(result)

    def wait(self, timeout=None):
        return True

    def get_future(self):
        return self._future

    def is_completed(self):
        return True

class MyBackend(dist.BackendBase):
    def __init__(self, store, rank, world_size, timeout):
        super().__init__(rank, world_size)
        self.num_allreduces = 0

    def getBackendName(self):
        return "my_backend"

    def allreduce(self, tensors, opts):
        # A real backend communicates here; with one rank, SUM is a no-op.
        self.num_allreduces += 1
        return MyWork(tensors)

dist.Backend.register_backend("my_backend", MyBackend, devices=["cpu"])
dist.init_process_group("my_backend", store=dist.HashStore(), rank=0, world_size=1)
t = torch.ones(2)
dist.all_reduce(t)
print(t)
print(dist.get_backend_impl().num_allreduces)
dist.destroy_process_group()
```

Functional collectives support backward()

- Functional collectives:
traceable, compilable dist.*
API
- Now differentiable: gradients
flow through backward
- Stable in torch 2.16

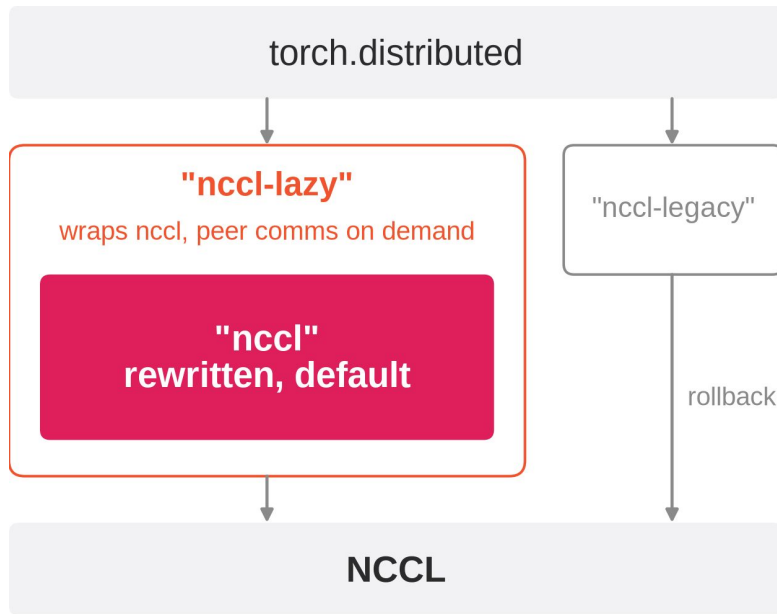
```
from torch.distributed import (  
    functional_collectives as funcol  
)  
  
t = torch.rand(10, device="cuda",  
    requires_grad=True)  
out = funcol.all_reduce(t, "sum", pg)  
out.sum().backward()  
assert t.grad is not None
```



NCCL Improvements

A rewritten NCCL backend

- Upstreamed TorchComms backend
- Legacy implementation had significant tech debt
- Single communicator per group
- New features
 - reconfigure, windows
- Validated: TorchTitan loss, grad norm, performance and memory usage
- Validated: FSDP, TP, PP, CP, HSDP, MoE



Behavior changes

	Legacy	Rewritten
Communicator init	Initializes on first collective	Eagerly initializes by default, lazy_init=True defers init
P2P / pipeline	Create peer communicators on demand	Single communicator "nccl-lazy" preserves behavior; PP warns
Devices	Multiple devices	Single communicator/device
Flight Recorder, NaN check	Built into the backend	Hooks; dump on timeout
Reconfigure, Windows	no support	Supported

Choosing the backend

- No code change: "nccl" selects it
- "nccl-lazy": peer communicators on demand
- "nccl-legacy": previous backend, removed in 2.16
- Pipeline parallel warns once if eager

```
# default: rewritten backend  
dist.init_process_group("nccl")
```

```
# peer communicators on demand  
dist.init_process_group("nccl-lazy")
```

```
# previous backend  
dist.init_process_group("nccl-legacy")
```

```
# lazy init on the rewritten backend  
opts = dist.ProcessGroupNCCL.Options()  
opts.lazy_init = True  
dist.init_process_group("nccl", pg_options=opts)
```

Collective Configs

- Allow for fine grained control on a per collective basis
- NCCL ≥ 2.31 + nccl4py ≥ 0.5
- nccl-legacy and gloo ignore it

```
# typed, via nccl4py >= 0.5
from nccl.core import NCCLCollConfig

cfg = NCCLCollConfig(max_ctas=4,
                     alg_selection="ring")
dist.all_reduce(t, config=cfg)

# or a plain dict
dist.all_reduce(t, config={
    "max_ctas": 4, "alg_selection": "ring"
})
```



Fault Tolerance

torch.distributed._reconfigure

- Allows for recovering from failures
- Scale up/down at runtime
- Dynamic world size
- Takes 2 inputs:
 - UUID to avoid split brain conditions
 - "reconfigure handle": unique metadata per worker from `dist._get_reconfigure_handle()`

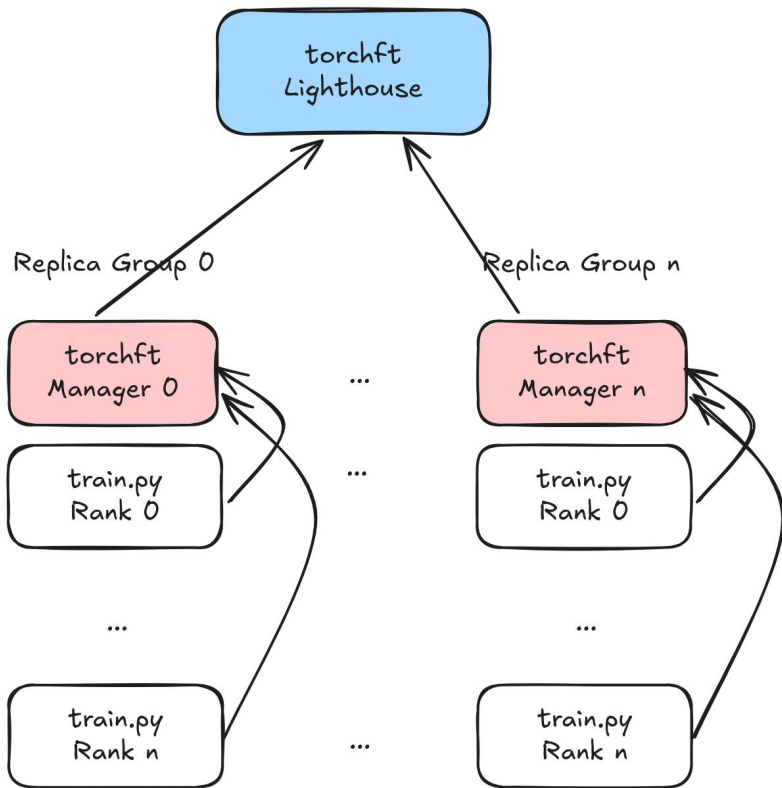
```
dist.init_process_group(  
    "nccl", enable_reconfigure=True)  
  
# publish handle out of band  
h = dist._get_reconfigure_handle()  
handles: list[str] = my_exchange(h)  
  
# survivors rebuild in place  
dist._reconfigure(  
    uuid=uuid, handles=handles,  
).wait()  
  
# new rank/world size  
dist.get_rank
```

Backend support

- Gloo: reconfigure
- NCCL:
 - new APIs `ncclCommRevoke`, `ncclCommShrink`, `ncclCommGrow`
- Validated on 256 GPUs: TorchTitan + torchft (FT HSDP, FT FSDP)
- MCCL: Meta's collective library, talk at this conference

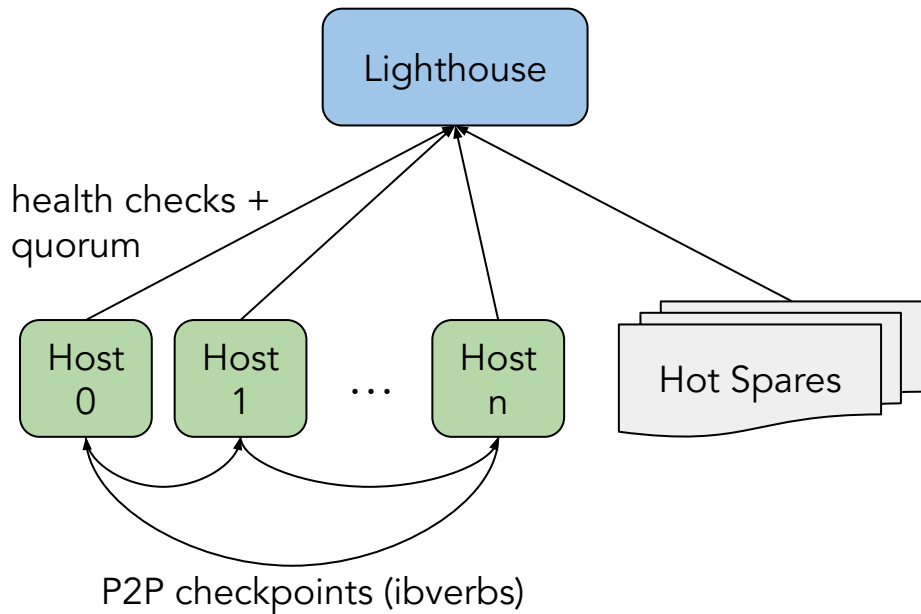
Previously: FT HSDP (replicated)

- torchft previously implemented FT HSDP
- Uses "replica groups" to isolate failures to part of the training job.
- Uses a "Lighthouse" to handle health checks + quorum on every training step



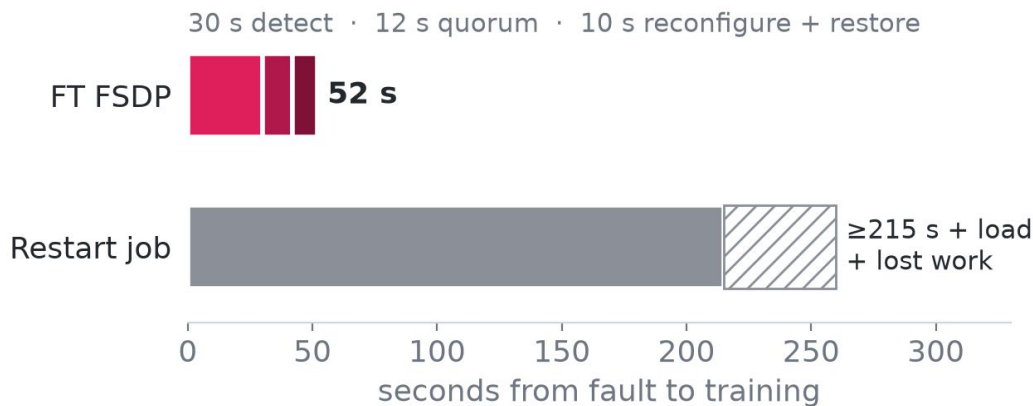
New: FT FSDP (sharded)

- Prototype
- Combines all the new features
 - Reconfigure – reconfiguration after failure
 - Transport – lightweight fast P2P checkpointing over ibverbs
- Checkpoint every step
- On error: swap in a replacement worker
- Uses torchft lighthouse for failure detection and quorum



FT FSDP: 256 H100, Llama3 8B

- 32 active + 2 spare hosts; 1 host killed every 10 min
- Spare fetches its shards over ibverbs in ~30 ms
- Async checkpoint: 66 ms copy + 34 ms replicate, hidden under a 1.26 s step

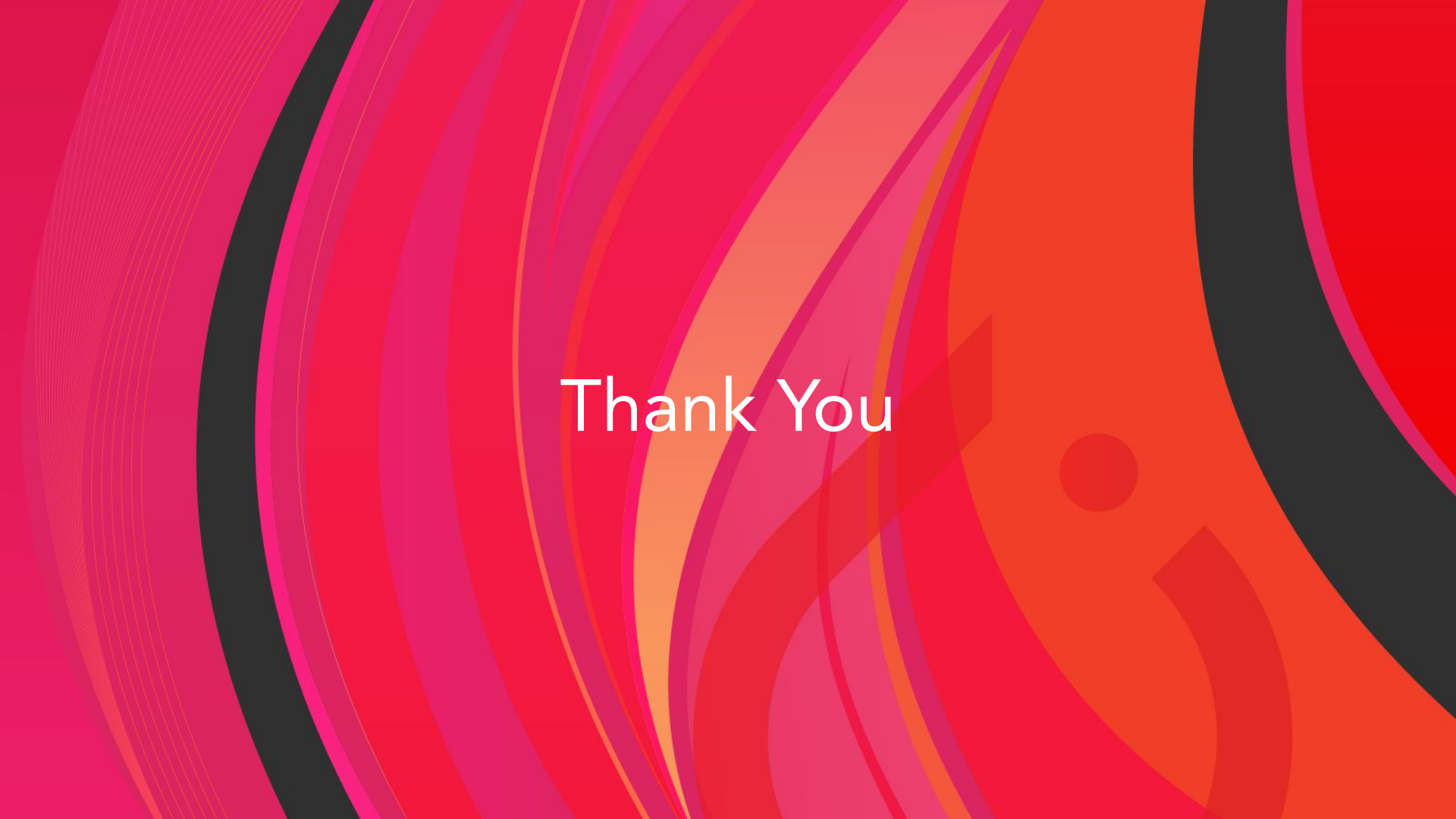


Thank you

- Next: *Beyond Collectives* (Symmetric Memory), Oct 21 11:45 AM, Room 210B
- pytorch.org/docs/main/distributed.html
- github.com/meta-pytorch/torchcomms
- github.com/meta-pytorch/torchft



Questions?



Thank You

API cleanups: `_single` naming

- Renamed to `_single`
- Coalesced: `*_single_coalesced`
- C++ ProcessGroup / Backend match
- Public `dist.set_timeout`

```
dist.all_gather_single(out, inp)
# was all_gather_into_tensor
dist.reduce_scatter_single(out, inp)
# was reduce_scatter_tensor
dist.gather_single(inp, out, dst=0)
# was gather_into_tensor
pg.all_gather_single(out, inp)
# was _allgather_base
```

Security Improvements: weights_only=True

- Object collectives accept `weights_only=True`
- Same restricted unpickler as `torch.load`
- Custom types: allowlist with `safe_globals`
- Not a security boundary: trusted networks only

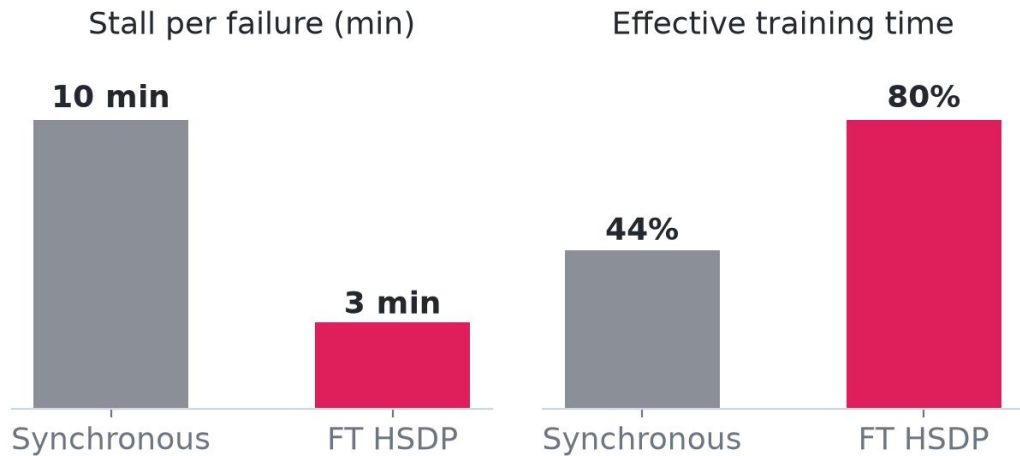
```
from torch.serialization import safe_globals
class MyData(NamedTuple):
    foo: torch.Tensor
obj = MyData(foo=torch.rand(10))
objs = [None] * dist.get_world_size()
# raises: MyData is not allowlisted
dist.all_gather_object(
    objs, obj, weights_only=True)
with safe_globals([MyData]):
    dist.all_gather_object(
        objs, obj, weights_only=True)
```

Takeaways

- Zero-copy RDMA: Windows (in-group) and Transports (any peer)
- Transfers capture into CUDA graphs
- Observe and extend every collective with hooks
- Ship backends as wheels via entry points
- Consistent `_single` collective names; old names kept
- Python backends; functional collectives backward; `weights_only`
- Rewritten NCCL backend by default; "nccl-legacy" to roll back
- Per-collective NCCL configs
- Survive failures without restarting: reconfigure
- Try it on PyTorch nightly — feedback welcome

FT HSDP at 98k GPUs

- Stall per failure: 10 min $\rightarrow$ 3 min
- Effective training time: 44% $\rightarrow$ 80%
- Simulated on 98,304 H100s (arXiv 2602.00277)



One failure shouldn't restart the job

- Today: any rank failure → abort → reschedule → reload checkpoint
- Restart cost scales with job size
- Goal: drop the failed ranks, keep training, heal later

Restart the job



Reconfigure in place



healthy ranks keep training

time