PyTorch RPC: Distributed Deep Learning Built on Tensor-Optimized Remote Procedure Calls

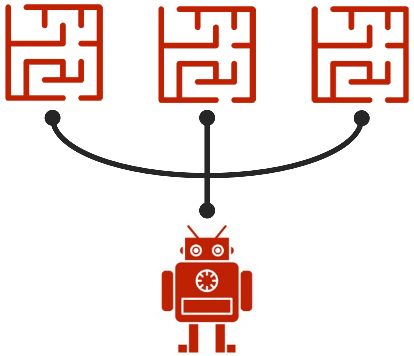
Shen L. Pritam D. Luca W. Rohan V. Omkar S. Pavel B. Howard H. Yanli Z.
Lucas H. Wanchao L. Hongyi J. Shihao X. Satendra G. Alisson A. Guoqiang C.
Zachary D. Chaoyang H.* Amir Z.* Alban D. Edward Y. Gregory C. Brian V.
Manoj K. Joe S. Salman A.* Soumith C.

Meta AI *University of Southern California

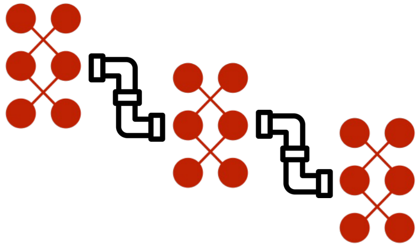
Motivation

- PyTorch DDP and FSDP are supports synchronized Single-Program Multi-Data (SPMD)

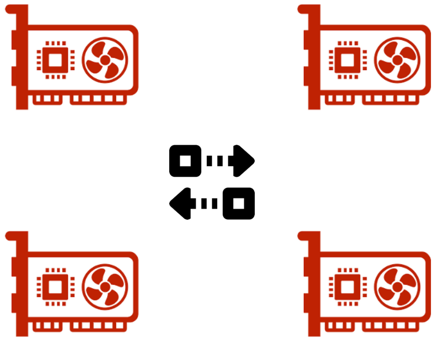
- Some applications do not fit well with the collective-based paradigms



1-to-many (e.g., RL)



pipeline (e.g., LLM)



async (e.g., GenAI)

- Can we build a generic low-level API to serve them all?

Motivation

- Existing options

Send/Recv from *CCL

- Pro: high-perf
- Con: Huge context on developers' shoulder
  - pickling
  - comm order & content
  - overlapping
  - etc.

3rd-party RPC

- Pro: great UX, except bwd
- Con: relatively low-perf due to the barrier between ML framework and comm layer
  - no CUDA-CPU overlap
  - no PCIe-TPC overlap
  - Sync on Tensor malloc
  - format conversion
  - etc.

- We want advantages from both sides

*CCL's perf + RPC's UX ⟶ PyTorch RPC

Programming Interface

`init_rpc()` on the process and give it a name

Run arbitrary functions including bulitin ones remotely and get the result back asynchronously

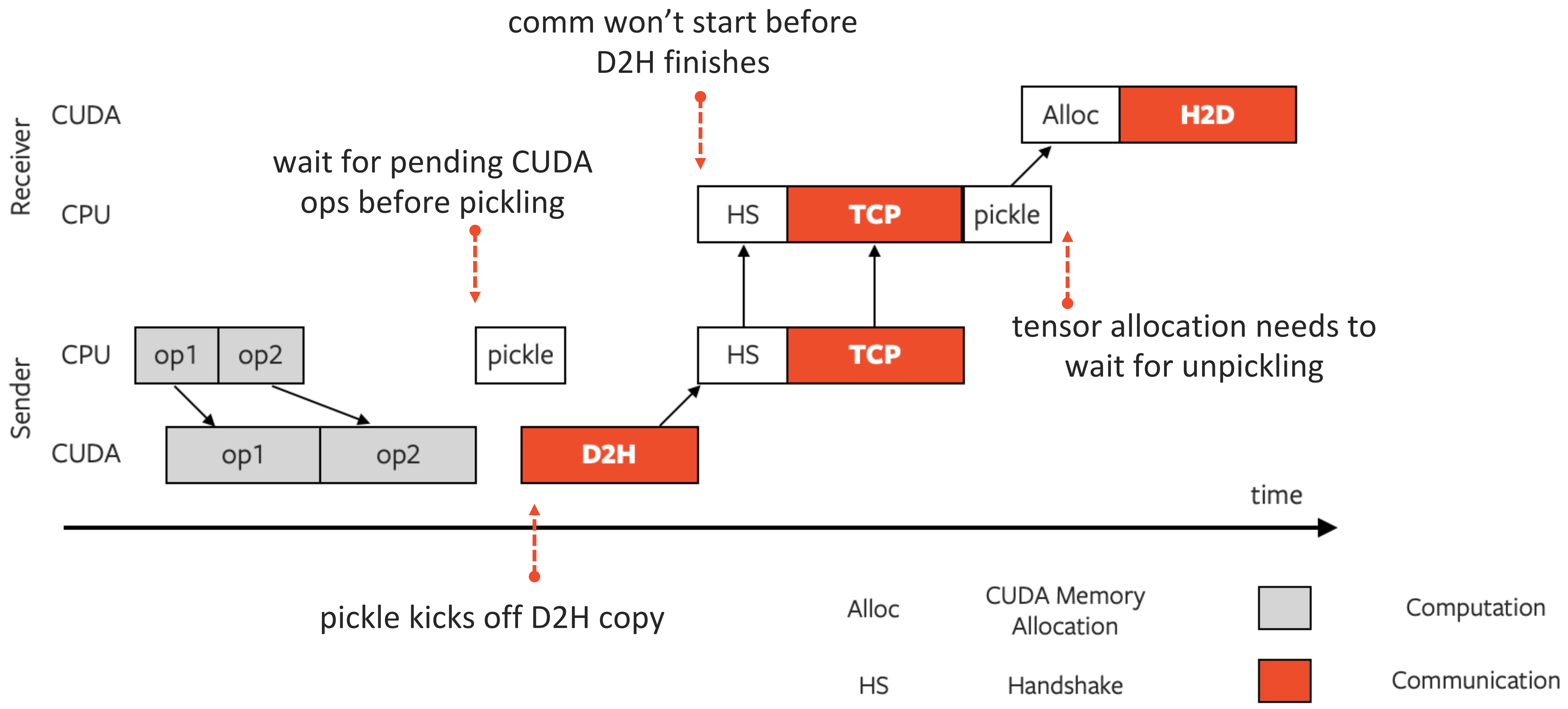
`remote()` keeps the result on the remote process and pass the reference around.

`to_here()` fetches the referenced data to the local process

call `backward()` as-if this is local training

```
# initialize RPC agent for this process
rpc_init("p0",…)

def my_add(x, y):
  return x + y

x = torch.zeros(2, requires_grad=True)

# async, returns future
fut = rpc_async("p1", my_add, args=(x, 1))

# async, returns reference of result
rref = remote(
  "p2", torch.add, args=(x, fut.wait())
)

# bwd will prop thru proc boundaries
rref.to_here().sum().backward()

# shutdown RPC agent
shutdown()
```

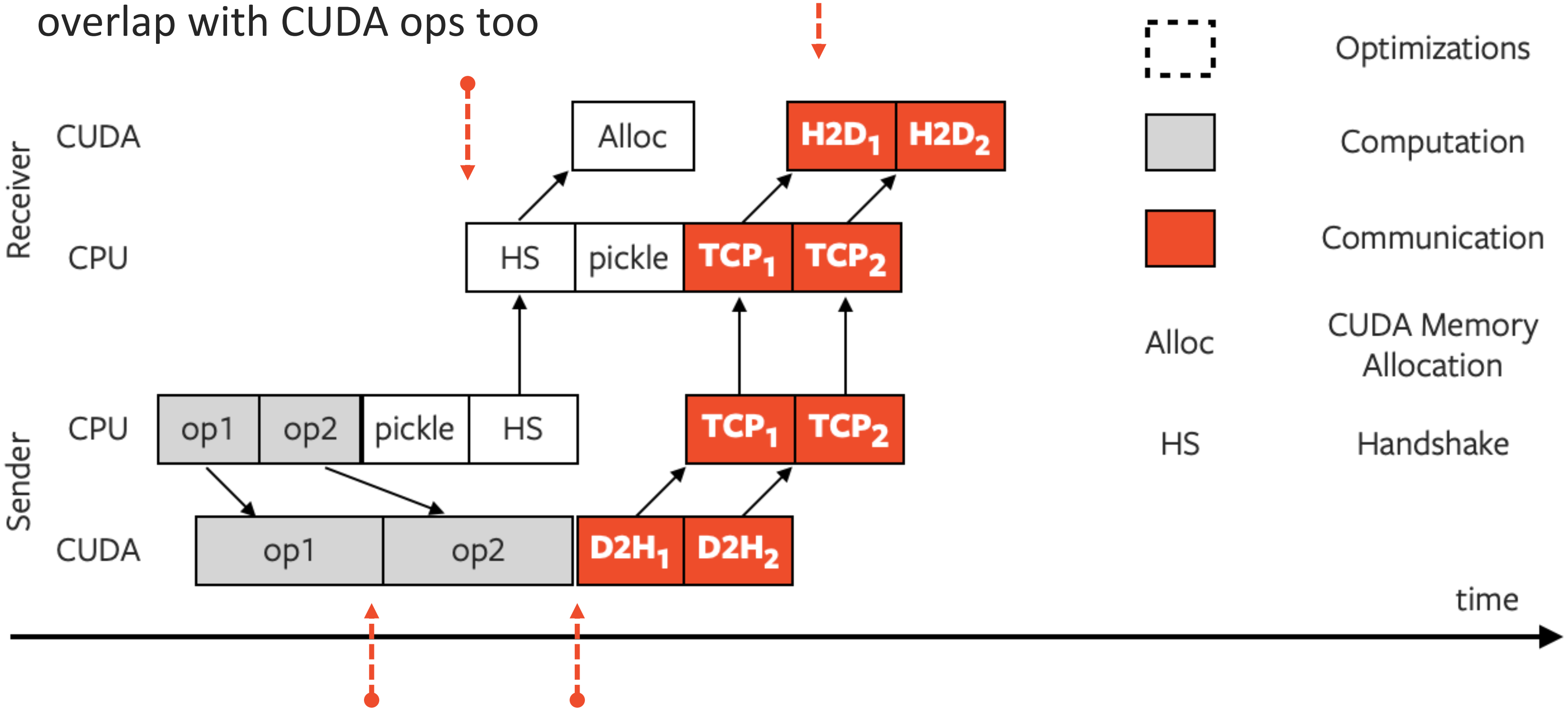
Tensor Communication with 3rd Party RPC



Optimized Tensor Communication

handshake uses CPU, can overlap with CUDA ops too

Tensors are sliced into chunks. Use staging buffer to overlap D2H, TCP, and H2D comms

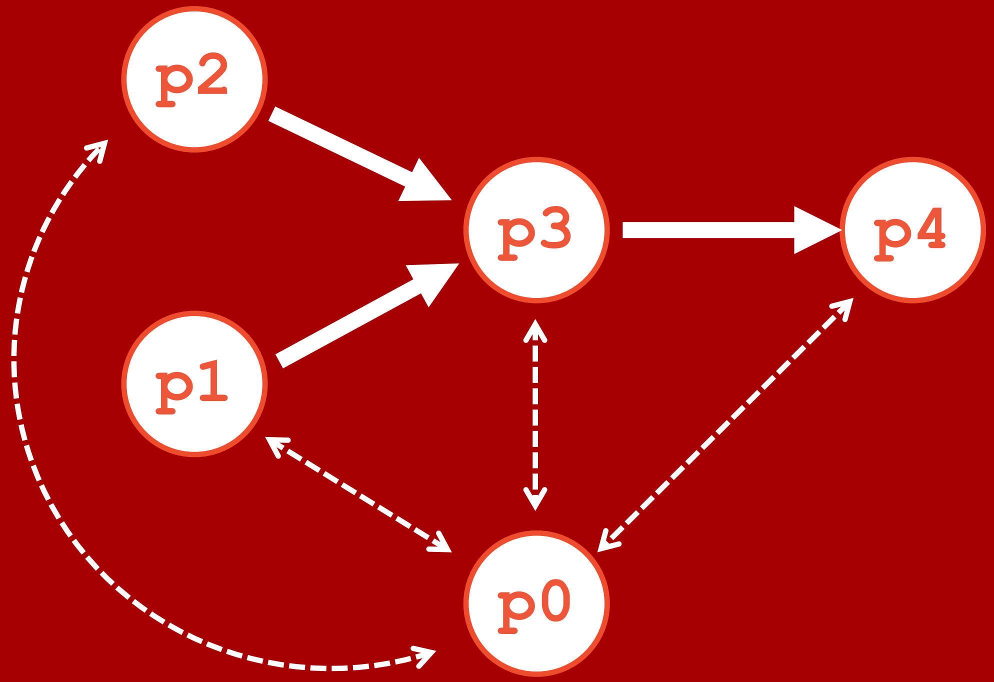


metadata pickling on CPU overlaps with pending CUDA ops

receiver can allocate tensors after recv metadata, which can overlap with CUDA ops, D2H comms and TCP comms

Remote Reference (RRef)

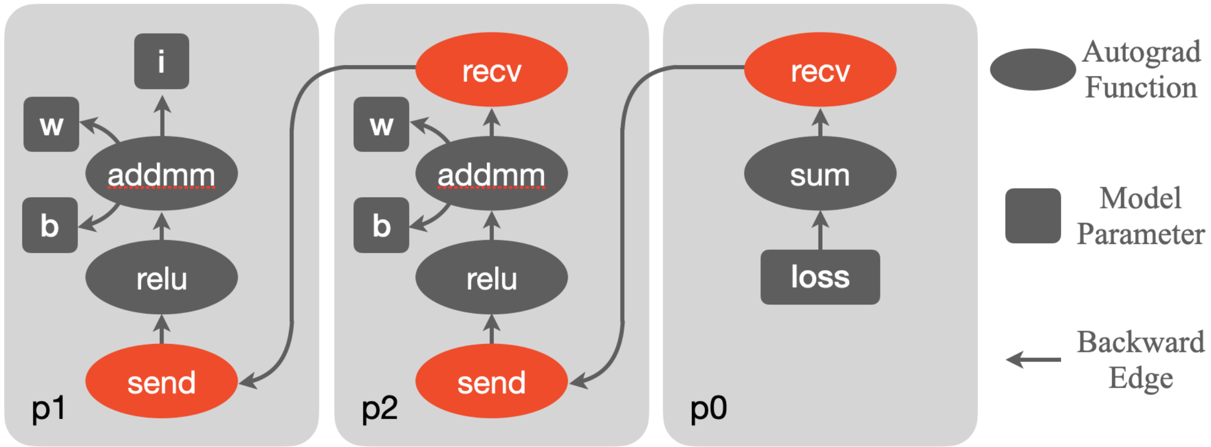
- RRef is “roughly” a distributed shared pointer
- Owner is the RRef that holds the data
- Owner RRef keeps reference counts and runs GC based accordingly.
- Owner RRef will be notified when a user RRef is forked or deleted.
- RRef allows separate out control plane with data plane




```
def rref_add(ra, rb):
  return ra.to_here() + rb.to_here()

# on worker process "p0"
ra = remote("p1", load_data_a)
rb = remote("p2", load_data_b)
rc = remote("p3", rref_add, args=(ra, rb))
rd = remote("p4", rref_add, args=(rc, rc))
```

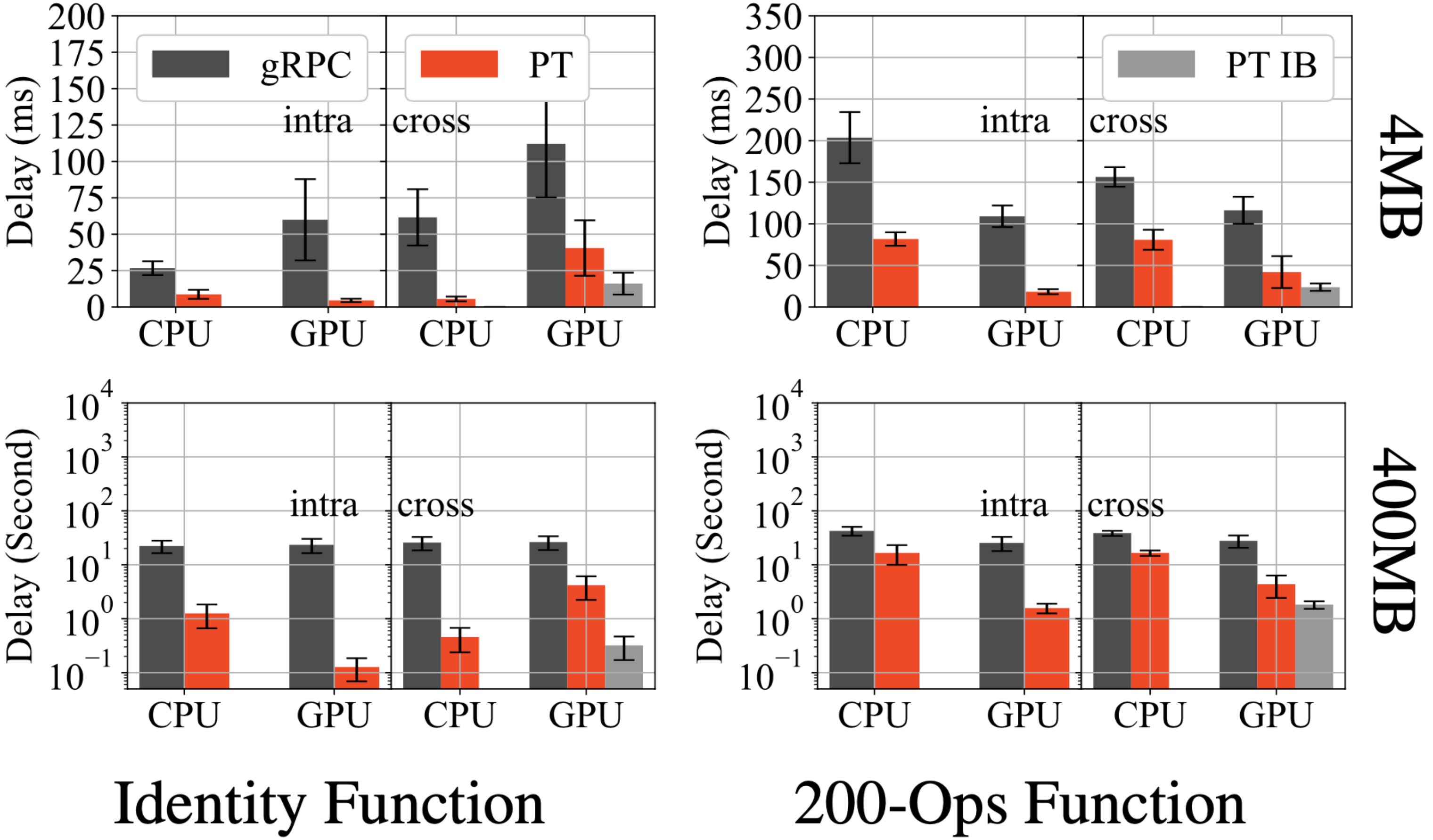
Distributed Autograd

- Each `rpc_async` and `to_here` call in forward installs a pair of `send`/`recv` autograd functions to connect local autograd graphs.
- `recv` recursively waits for its `send` peer during the backward propagation.
- Gradients are stored in dedicated distributed context for every concurrent backward.




```
# rx, ri, and ry are a RRef
class Block(nn.Module):
  def forward(self, rx):
    # .to_here() triggers tensor comm
    return self.relu(self.fc(rx.to_here()))

class Model(nn.Module):
  def __init__(self):
    self.rb1 = remote("p1", Block)
    self.rb2 = remote("p2", Block)

  # run forward on "p0"
  def forward(self, ri):
    rx = self.rb1.remote().forward(ri)
    ry = self.rb2.remote().forward(rx)
    return ry

m = Model()
loss = m(RRef(i)).to_here().sum()
```

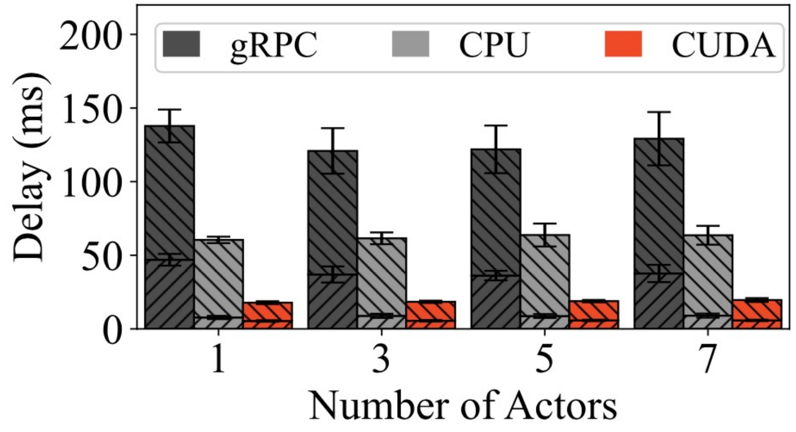
Latency Comparison



- Tensor {4MB, 400MB} X func {Id, 200ops}
- CPU Intra-node: SHM/CMA
- CPU Inter-node: 4X100Gbps Ethernet
- GPU Intra-node: NVLink, PCIe
- GPU Inter-node: 4X100Gbps Ethernet, IB
- Every call repeated 10 times
- Largest lead observed in 400MB + Id + GPU

- Speedups
  - Direct access to Tensor storage
  - Multiple types of overlap
  - Diverse HW-related optimizations

Reinforcement Learning



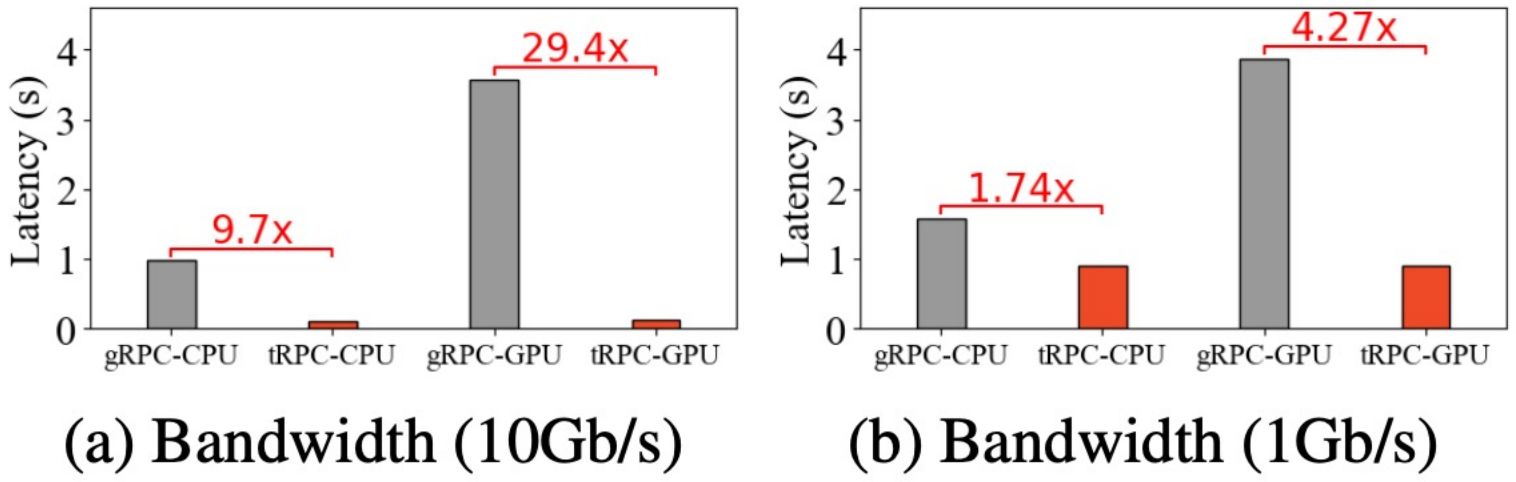
	gRPC	CPU	CUDA
LoC Comm LoC Comp	52 (+270 gen) 258	10 258	12 258
Rewards Max=700	654.3 ± 104.0	676.3 ± 52.03	674.0 ± 83.83

- Env: OpenAI Gym
- Algorithm: Async Advantage Actor Critic (A3C)
- Actor behavior:
  - Fetch global network
  - Interact with env
  - Generate grads
  - Push grads to update global network

- Enable PyTorch RPC only requires ~10 LoC

Federated Learning

- Integrated with fedml.ai
- FL Clients use multiple AWS accounts but share the same AWS region
- Server: AWS EC2 p3.2xlarge (8 CPU cores, 1 V100 GPU)
- Training 25M-param model with FedAvg and FedSGD



Thanks!

- Github: https://github.com/pytorch/pytorch/blob/main/torch/distributed/rpc
- Docments: https://pytorch.org/docs/stable/rpc.html
- Forum: https://discuss.pytorch.org/c/distributed/12
- Slack: https://pytorch.slack.com/archives/CBHSWPNM7