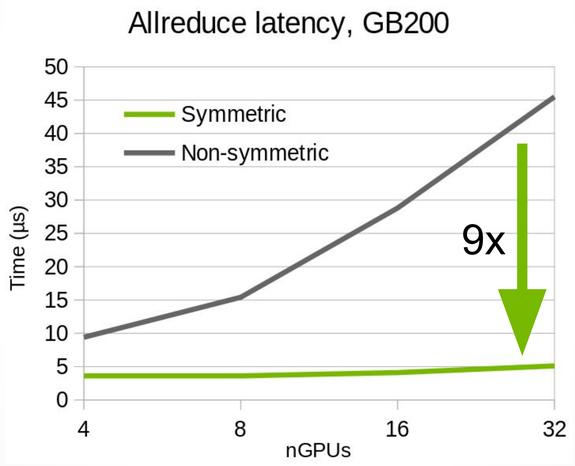
PyTorch Symmetric Memory + NCCL Device APIs: A New Path Towards Multi-GPU Kernels

Ke Wen, Sylvain Jeaugey

PyTorch Conference Europe, April 2026

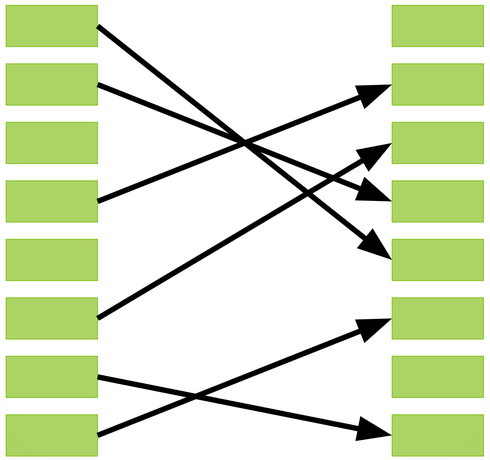
Challenges in Today's Distributed Workloads

Latency-sensitive operations
Inference



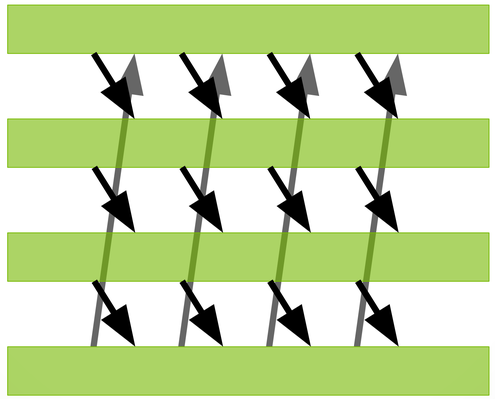
vLLM, SGLang, TRT-LLM

Specific comm patterns
E.g. MoE



On-device metadata (dest IDs, splits)

Comm/compute fusion
In-kernel communication

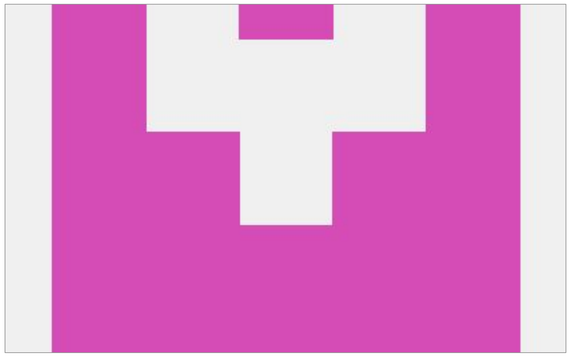


Flash-infer, Helion

- What is symmetric memory?
- NCCL device APIs
- NCCL EP
- PyTorch Symm Mem features

Disclaimer

PyTorch Symmetric Memory is developed by open-source community



Yifu Wang

Ke Wen

Natalia Gimelshein

Aaron Gokaslan

Chien-Chin Huang

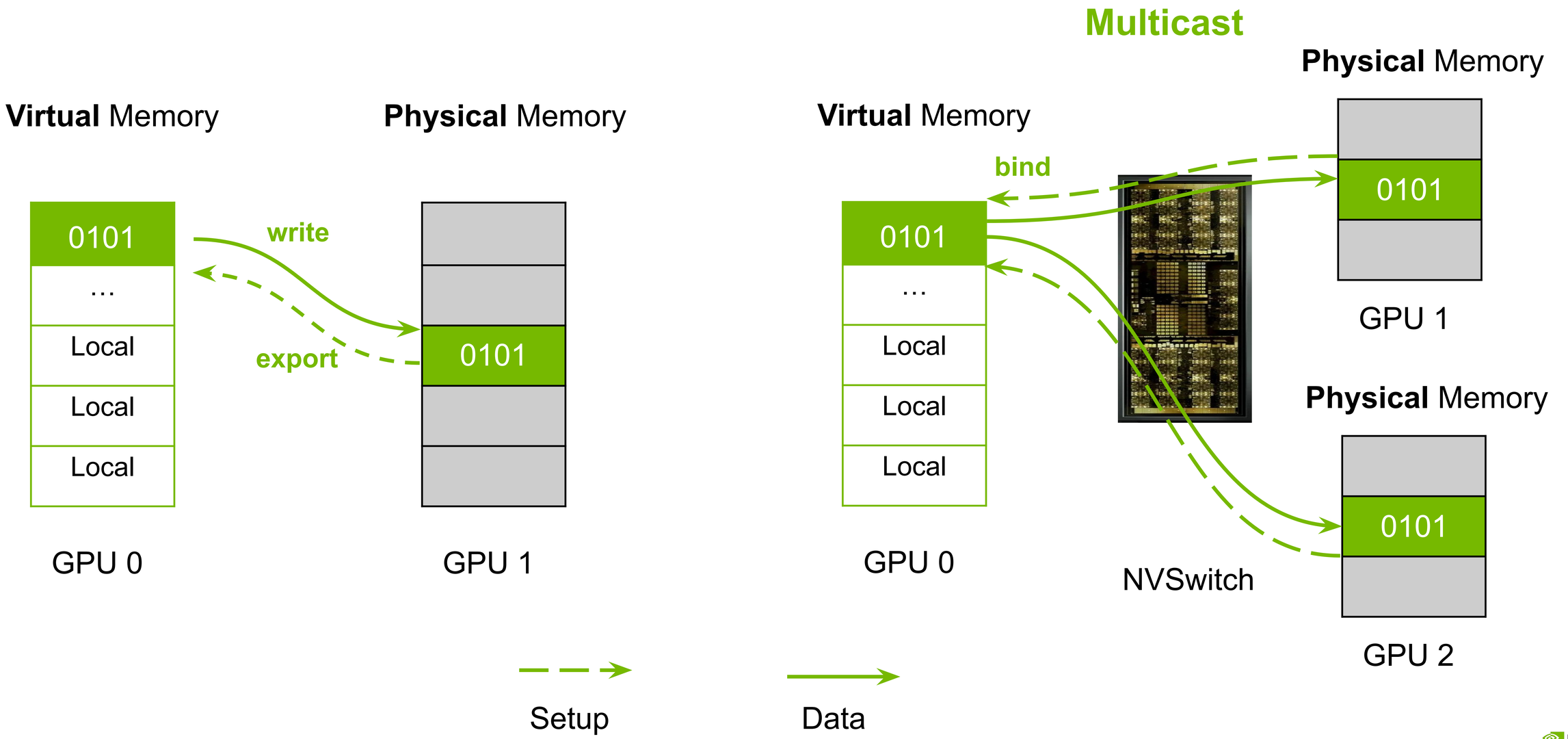
Junjie Wang

Prachi Gupta

and many others

What is Symmetric Memory

A simplified view



PyTorch Workflow

A “hello world” example

- **Import module**

```
import torch.distributed._symmetric_memory
as symm_mem
```

- **Creating Symmetric Tensors**

```
t = symm_mem.empty(128, device=”cuda:0”)
hdl = symm_mem.rendezvous(t, group)
```

- **Calling Collective Ops**

```
torch.ops.symm_mem.one_shot_all_reduce(
    t, “sum”, group
)
```

Host

- **Your Kernel**

```
__global__ void kernel(...) {
  // Sync with remote peer(s)
  ...
  // Fetch data from peer
  ...
  // Local compute
  ...
  // Write result to local or remote mem
  ...
}
```

Device

NCCL Device APIs

NCCL Device API

In-kernel communication toolbox

Host-side API

Device Communicator

Designates a group of ranks

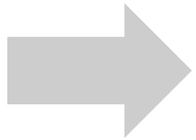
Device-side equivalent to the NCCL "comm"
Created with ncclDevCommCreate.

Window

Designates memory buffers

Group of buffers registered across all ranks.
Created with ncclCommWindowRegister.

passed to the kernel



Device-side API

LSA	Communication primitives for **Load/Store Accessible** memory
Multimem	Communication primitives for **NVLink SHARP**
GIN	**GPU-Initiated Networking** primitives

NCCL Device API

In-kernel communication toolbox

LSA

Load/Store Accessible

Remote memory access
→Get pointer for remote memory
→Perform normal memory operations to pointer

Synchronization
→ LSA barrier

Multimem

NVLink SHARP

NVLink SHARP access
→Get pointer to multimem mappings
→ Perform multimem operations on pointer

GIN

Gpu-Initiated Networking

Remote memory access
→Create GIN object, select GIN context
→Perform put() operations
→Wait on signals for remote operations
→Wait on counters for local operations
Synchronization
→GIN barrier

```
__global__ kernel(ncclDevComm_t devComm, ncclWindow_t win) {

  float* ptr = ncclGetPeerPointer(devComm, offset, peer);
  ptr[0] = value;

  ncclBarrierSession bar {
      ncclCoopCta(), ncclTeamTagLsa(), devComm, blockIdx.x };
  bar.sync(ncclCoopCta(), cuda::memory_order_release);

  float* ptr = ncclGetLsaMultimemPointer(win, offset, devComm);
  value = multimem_reduce_sum(ptr);
  multimem_store(ptr, value);

  ncclGin gin { devComm, blockIdx.x % devComm.ginContextCount };
  gin.put(ncclTeamWorld(devComm), peer,
      win, remoteOffset, win, localOffset,
      size, ncclGin_SignalInc{signalIndex});
  gin.waitSignal(ncclCoopCta(), signalIndex, signalValue);

  ncclBarrierSession bar {
      ncclCoopCta(), ncclTeamTagWorld(), gin, blockIdx.x };
  bar.sync(ncclCoopCta(), cuda::memory_order_relaxed,
      ncclGinFenceLevel::relaxed);

}
```

NCCL Device API

In-kernel communication toolbox

LSA

Load/Store Accessible

Remote memory access
→Get pointer for remote memory
→Perform normal memory operations to pointer

Synchronization
→ LSA barrier

Multimem

NVLink SHARP

NVLink SHARP access
→Get pointer to multimem mappings
→ Perform multimem operations on pointer

GIN

Gpu-Initiated Networking

Remote memory access
→Create GIN object, select GIN context
→Perform put() operations
→Wait on signals for remote operations
→Wait on counters for local operations
Synchronization
→GIN barrier

```
__global__ kernel(ncclDevComm_t devComm, ncclWindow_t win) {

  float* ptr = ncclGetPeerPointer(devComm, offset, peer);
  ptr[0] = value;


  ncclBarrierSession bar {
      ncclCoopCta(), ncclTeamTagLsa(), devComm, blockIdx.x };
  bar.sync(ncclCoopCta(), cuda::memory_order_release);




  float* ptr = ncclGetLsaMultimemPointer(win, offset, devComm);
  value = multimem_reduce_sum(ptr);
  multimem_store(ptr, value);




  ncclGin gin { devComm, blockIdx.x % devComm.ginContextCount };
  gin.put(ncclTeamWorld(devComm), peer,
      win, remoteOffset, win, localOffset,
      size, ncclGin_SignalInc{signalIndex});
  gin.waitSignal(ncclCoopCta(), signalIndex, signalValue);


  ncclBarrierSession bar {
      ncclCoopCta(), ncclTeamTagWorld(), gin, blockIdx.x };
  bar.sync(ncclCoopCta(), cuda::memory_order_relaxed,
      ncclGinFenceLevel::relaxed);

}
```

NCCL Device API

In-kernel communication toolbox

LSA

Load/Store Accessible

Remote memory access
→Get pointer for remote memory
→Perform normal memory operations to pointer

Synchronization
→ LSA barrier

Multimem

NVLink SHARP

NVLink SHARP access
→Get pointer to multimem mappings
→ Perform multimem operations on pointer

GIN

Gpu-Initiated Networking

Remote memory access
→Create GIN object, select GIN context
→Perform put() operations
→Wait on signals for remote operations
→Wait on counters for local operations
Synchronization
→GIN barrier

```
__global__ kernel(ncclDevComm_t devComm, ncclWindow_t win) {

  float* ptr = ncclGetPeerPointer(devComm, offset, peer);
  ptr[0] = value;


  ncclBarrierSession bar {
      ncclCoopCta(), ncclTeamTagLsa(), devComm, blockIdx.x };
  bar.sync(ncclCoopCta(), cuda::memory_order_release);




  float* ptr = ncclGetLsaMultimemPointer(win, offset, devComm);
  value = multimem_reduce_sum(ptr);
  multimem_store(ptr, value);




  ncclGin gin { devComm, blockIdx.x % devComm.ginContextCount };
  gin.put(ncclTeamWorld(devComm), peer,
      win, remoteOffset, win, localOffset,
      size, ncclGin_SignalInc{signalIndex});
  gin.waitSignal(ncclCoopCta(), signalIndex, signalValue);


  ncclBarrierSession bar {
      ncclCoopCta(), ncclTeamTagWorld(), gin, blockIdx.x };
  bar.sync(ncclCoopCta(), cuda::memory_order_relaxed,
      ncclGinFenceLevel::relaxed);

}
```

CuTe DSL Support for NCCL

Enable fusion with compute

Compute / comm in one kernel

- Effectively “Zero SM”

Python level

- CuTe DSL

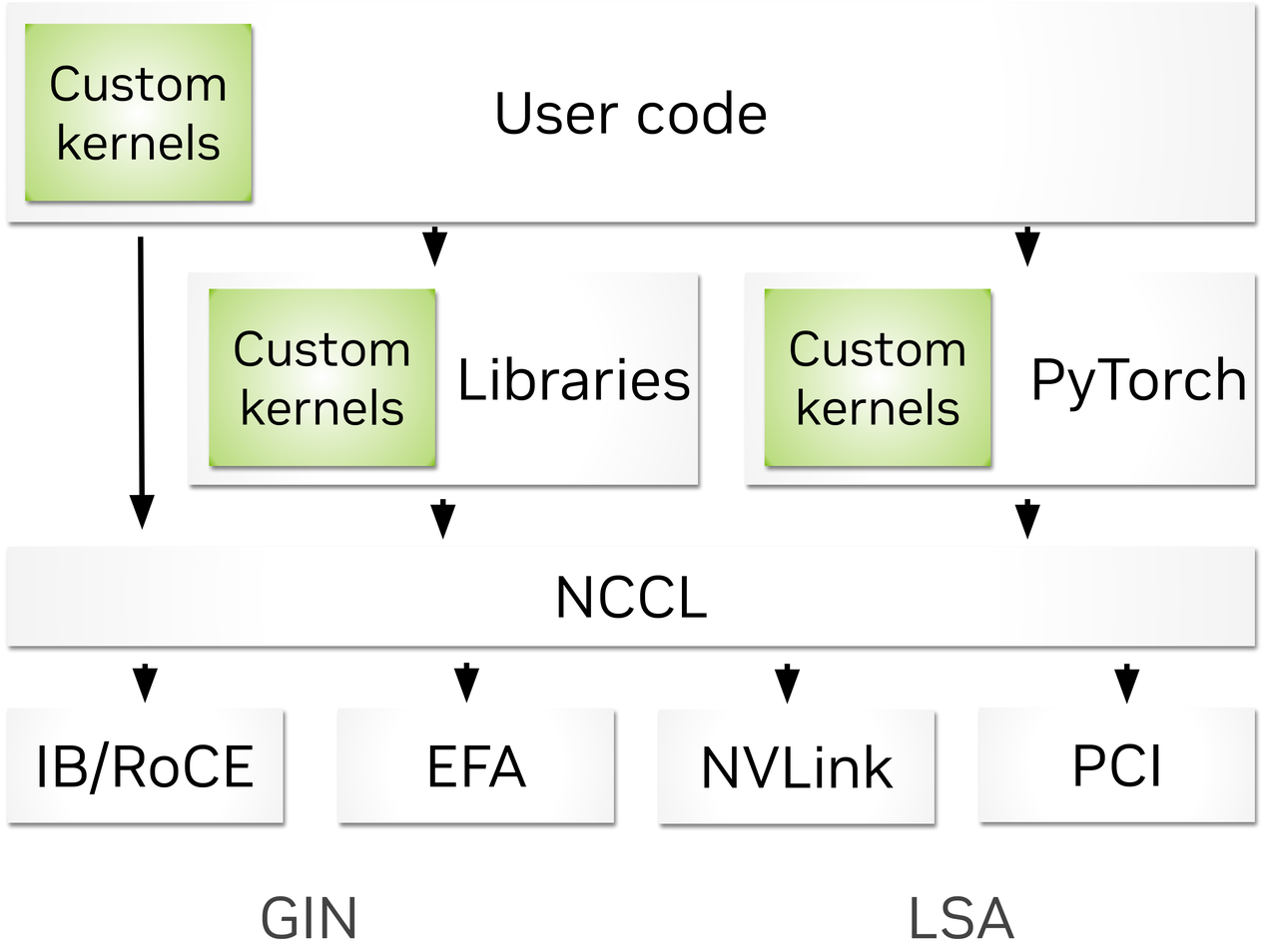
Auto JIT optimized

```python
import cutlass.cute as cute
from nccl.core.device.cute import core, gin, coop

@cute.kernel
def gin_put_kernel(dev_comm, window):
    team = core.team_world(dev_comm)
    gin_ctx = gin.init(dev_comm, gin.GinBackendMask.ALL, 0)
    coop_ctx = coop.init_cta()
    if team.rank == 0:
        gin.put(gin_ctx, coop_ctx, team, dst_rank, window, offset,
                signal_id=1, signal_op=0, ...)
    elif team.rank == 1:
        gin.wait_signal(gin_ctx, coop_ctx, signal=1, least=1)
```

Custom Communication Kernels

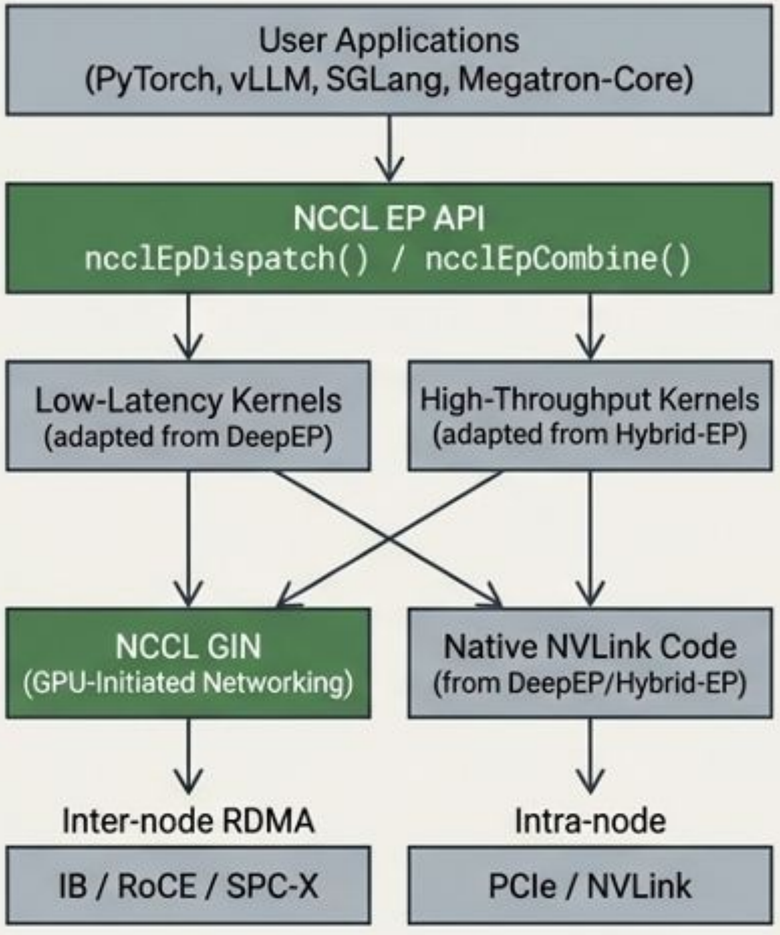
Integration stack



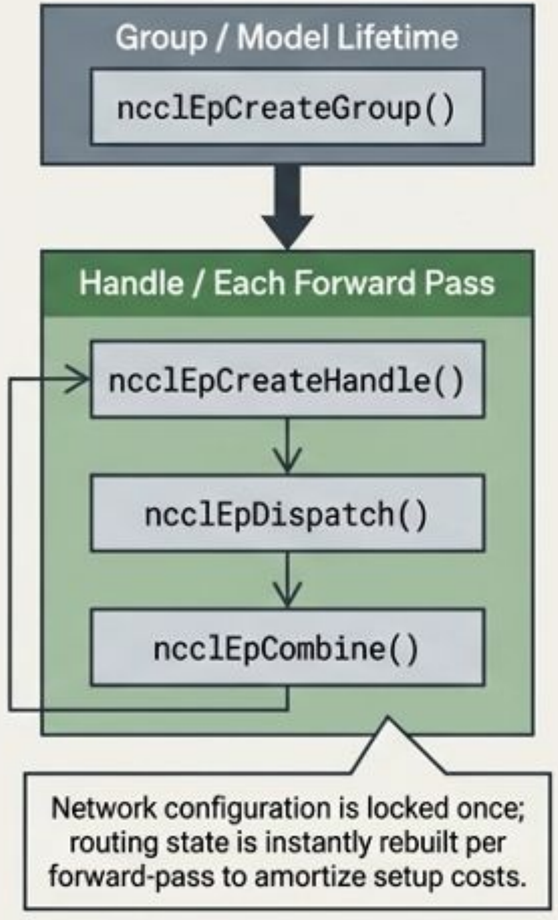
NCCL EP:
from device APIs to library

NCCL EP: Unified Architecture & Feature-Complete API

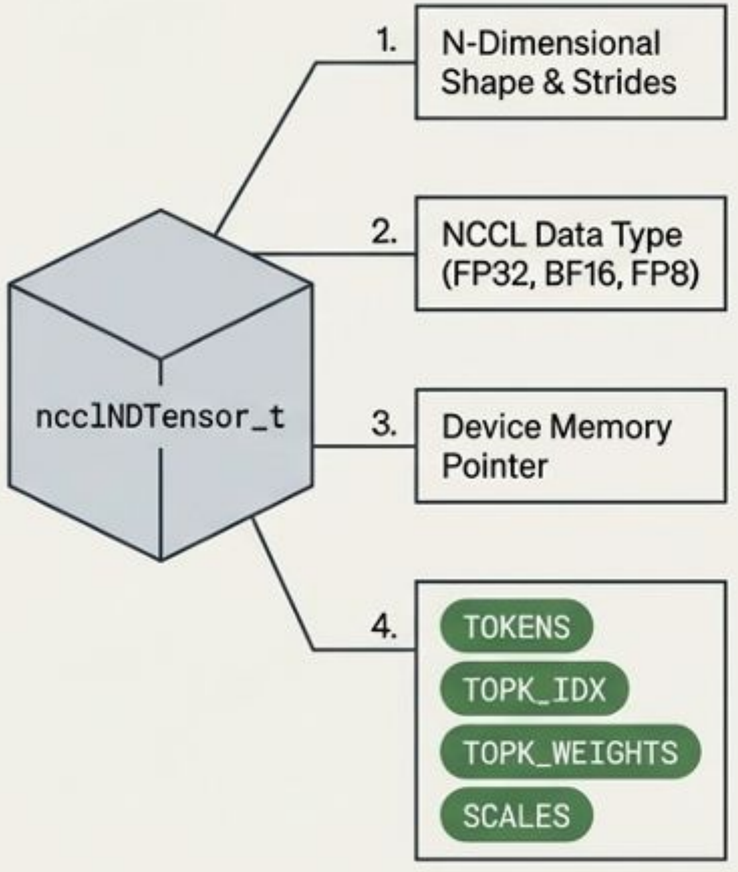
1. The Native Stack



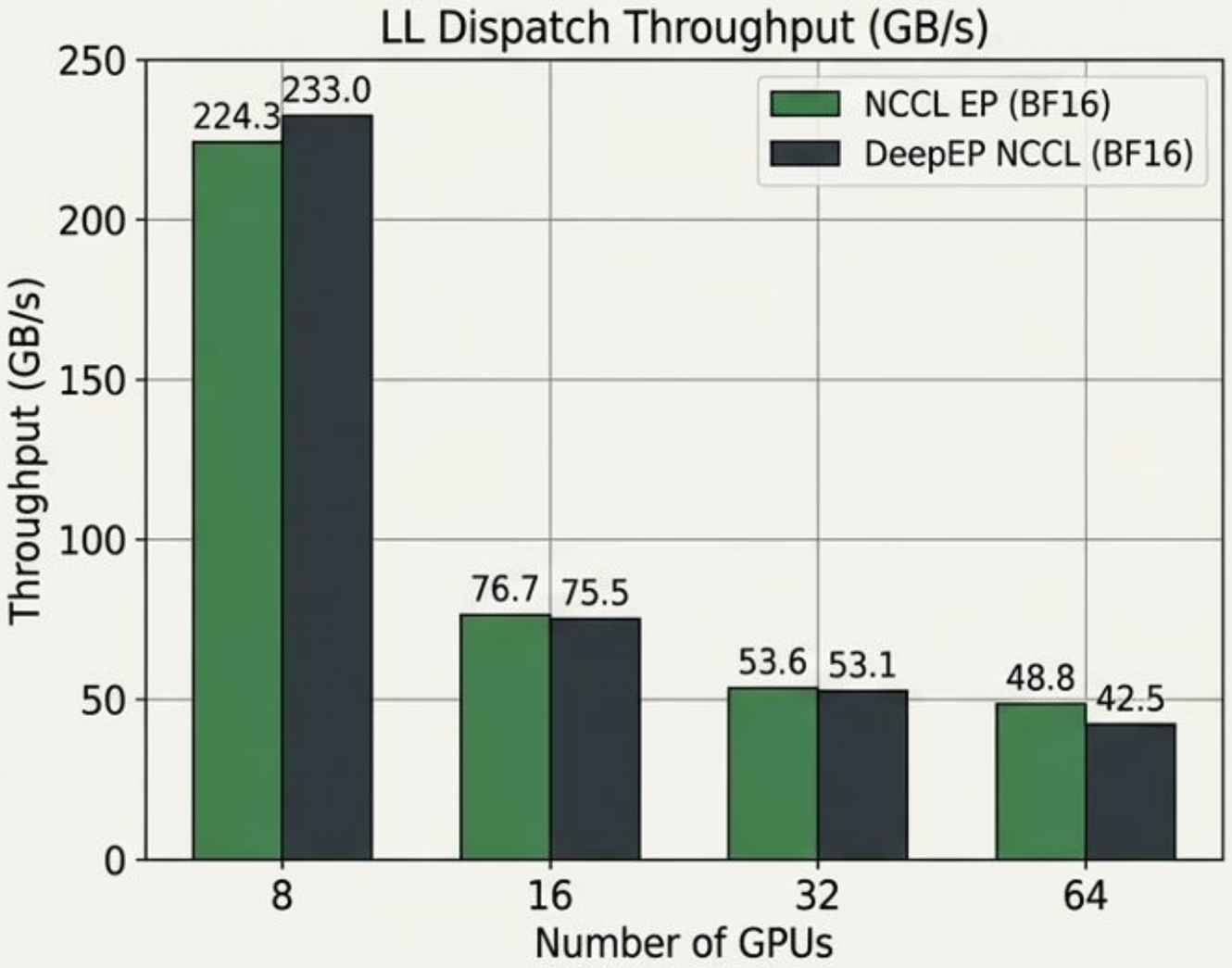
2. Two-Tiered Resource Management



3. Rich Tensor Semantics



Performance Parity: Matching State-of-the-Art Dispatch Throughput



NCCL EP matches or exceeds DeepEP at multi-node scales (2–8 nodes), achieving a notable throughput advantage at 64 GPUs (48.8 GB/s vs 42.5 GB/s).

Testing Context

```
Hardware: H100 EOS Cluster
Experts:  256
Tokens:   128
Routing:  top-k 8
```

NCCL EP Resources

Source code

- In NCCL repo, /contrib
- “GitHub-first” development
- PR and RFE welcome

Paper

NCCL EP: Towards a Unified Expert Parallel Communication API for NCCL

Integration (WIP)

- PyTorch
- TRT-LLM
- vLLM
- SGLang

```
torch.distributed.TokenSwitch


ts.bind_rounting(topk_idx)
ts.dispatch(...)
ts.combine(...)
```

PyTorch Symmetric Memory Features

Memory Pool Support

1. Default backing of MemPool:

(available from torch 2.11)

```
x = symm_mem.empty(1024)
symm_mem.rendezvous(x, group)
del x

# Reuse allocation, zero overhead
x = symm_mem.empty(1024)
symm_mem.rendezvous(x, group)
```

2. Pinning op output in Symm Mem:

```
mempool = symm_mem.get_mem_pool(device)
with torch.cuda.use_mem_pool(mempool):
     # y will be in symmetric memory
     y = torch.mm(x, w)
ops.symm_mem.one_shot_all_reduce(y)
```

3. torch.compile auto placement (WIP):

```
from torch.library import Library

lib = Library("my_lib", "DEF")
lib.define("foo(Tensor input) -> Tensor")
lib.register_symm_mem_args("foo", ["input"])
```

API reference

SymmMem Collectives

Low latency

`torch.ops.symm_mem.`

- `one_shot_all_reduce`
- `two_shot_all_reduce_`
- `multimem_all_reduce_`
- `multimem_all_gather_out`

One-sided

- `symm_mem.put_signal`
- `symm_mem.wait_signal`

Irregular patterns

`torch.ops.symm_mem.`

- `all_to_all_vdev`
- `all_to_all_vdev_2d (dispatch)`
- `all_to_all_vdev_2d_offset (combine)`
- `multi_root_tile_reduce (Shampoo + FSDP)`
- `reduce_scatter_offset (column wise)`
- `all_to_all_permute (Ulysses Parallel)`

Contribution is most welcome!

Copy Engine Collectives

Available in torch 2.11

```
cudaMemcpyBatchAsync(void** dsts, void** srcs, …)
```

SymmMem provides remote pointers

Enables:

- **All-gather**
- **All-to-all**

- **Frees CUDA SMs**
- **Less interference with computation**

```
# User code
opts.config.cta_policy = NCCL_CTA_POLICY_ZERO
dist.init_process_group(backend="nccl", pg_options=opts)

symm_mem.set_backend("NCCL")
inp = symm_mem.empty(numel, device)
out = symm_mem.empty(numel * world_size, device)
# Register tensors for symmetric memory operations
symm_mem.rendezvous(inp, group=group_name)
symm_mem.rendezvous(out, group=group_name)

# Same collective
work = dist.all_gather_into_tensor(out, inp, async_op=True)
```

PyTorch FSDP on Copy Engine

Available in torch 2.11

Micro benchmark:

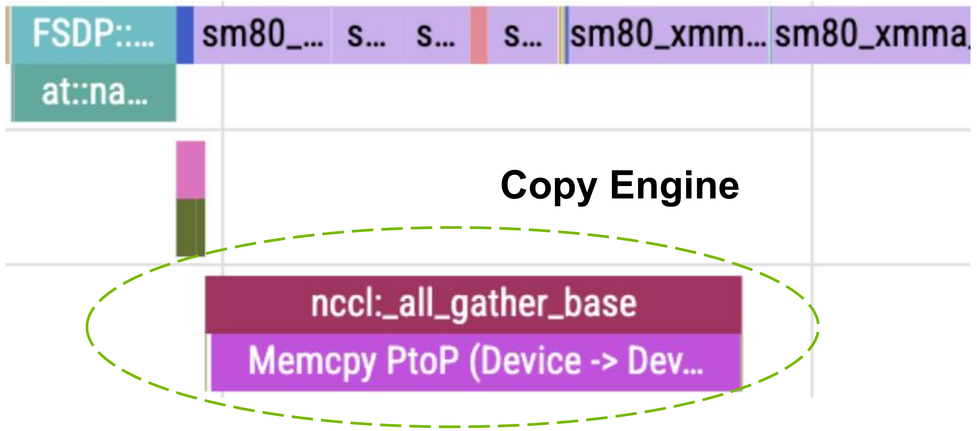
- Overlapping all-gather and matmul
- i.e. fetching next layer while computing current layer

	Time (ms)	Speedup
Without Copy Engine	**2.02**	**–**
With Copy Engine	**1.77**	**14%**

m = n = k = 8192, 8 x H100

User code:

```
>> model = Transformer(args)
>> fully_shard(model)
>> model.set_symm_mem_for_comm()
```




NVIDIA
Tell us what op you need
Contribute ops

Questions?