



CONFERENCE

—NORTH AMERICA 2026—

Beyond Collectives

Building Fused Compute-Communication Kernels with PyTorch Symmetric Memory

Natalia Gimelshein, Kapil Sharma (PyTorch, Meta)
Ke Wen (NVIDIA)

Beyond Collectives

Building Fused Compute-Communication Kernels with PyTorch Symmetric Memory

1

Recap

What is symmetric memory and composition with kernel DSLs

2

Community adoption

Ecosystem of kernels and libraries community has built

3

Symmetric memory pools

What if you wanted plain old dist.*

4

Case study: FlashInfer

Fused communication in inference kernels on symmetric memory: Decode context parallel + LSE fusion

5

Case study: CFT transport

CFT transport with symmetric memory (scaled-out kernel with multi-rack and counted put capability)



Contributors · Q&A

Who built it · other talks · how to reach us



Recap

Quick Recap – What is Symmetric Memory?

A tensor allocated on every rank with the same size and layout, then mapped so any GPU can read and write any peer's copy directly.

- **Symmetric:** Same size, layout and offsets on every rank
- **Mapped:** One rendezvous gives each GPU a handle to every peer's buffer
- **Device-initiated:** Kernels move data with plain loads and stores
- **Programmability:** Use the built-in ops, or fuse communication into your own kernel in Triton, CuTe DSL, Helion, CUDA, etc.

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

group = dist.group.WORLD
# 1. allocate: the same buffer on every rank
t = symm_mem.empty(4096, dtype=torch.bfloat16, device="cuda")

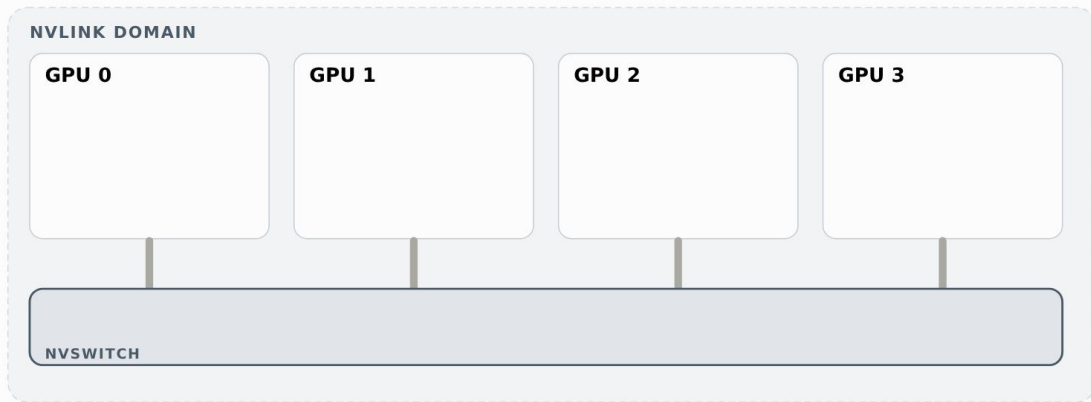
# 2. rendezvous: map every peer (+ multicast, if available)
hdl = symm_mem.rendezvous(t, group)

# 3a. a built-in op on the symmetric buffer ...
out = torch.ops.symm_mem.one_shot_all_reduce(t, "sum",
group.group_name)

# 3b. ... or raw addresses for your own kernel
my_kernel[grid](hdl.buffer_ptrs_dev, hdl.multicast_ptr,
                hdl.signal_pad_ptrs_dev)
```

Rendezvous

Every rank allocates some buffer. One collective rendezvous maps every peer to each other



handle

`buffer_ptrs[0..W-1]`

peer loads and stores

`multicast_ptr`

one store / reduce → every GPU

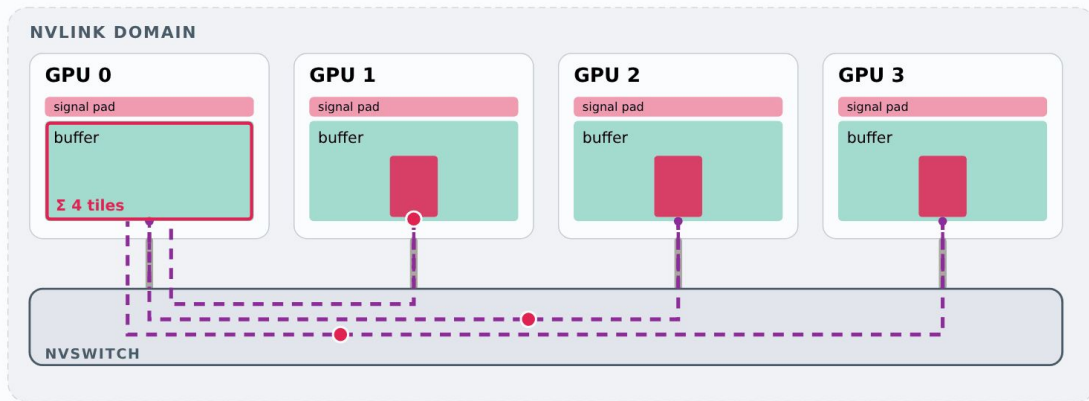
`signal_pad_ptrs`

barriers and signals

```
t = symm_mem.empty(n, device="cuda")      # same buffer + signal pad on every rank
hdl = symm_mem.rendezvous(t, group)        # exchange handles, map all peers (+ multicast)
hdl.buffer_ptrs  hdl.multicast_ptr  hdl.signal_pad_ptrs    # what kernels receive
```

Raw peer access: plain loads/stores

Successful rendezvous handle allows a kernel on GPU i reads data from every peer's buffer



- Peer address = `buffer_ptrs[r] + offset`
- Same layout on every rank
- Plain (or multicast) loads and stores over NVLink

```
peer = hdl.get_buffer(r, t.shape, t.dtype) # a tensor view of GPU r's copy of the buffer
x = peer[offs] # reading it is a plain load from GPU r's memory
acc = sum(hdl.get_buffer(r, t.shape, t.dtype)[offs] for r in range(world)) # = one-shot all-reduce
```

Under the hood: Rendezvous Machinery

3 backends – started with CUDA, later nvshmem and now NCCL is supported as first-class backend

STEP	CUDA BACKEND (DEFAULT)	NCCL BACKEND
Allocate	cuMemGetAllocationGranularity → cuMemCreate cuMemAddressReserve → cuMemMap → cuMemSetAccess	ncclMemAlloc
Exchange	cuMemExportToShareableHandle (POSIX file descriptor, or fabric handle on NVL72) FDs over a UNIX socket; metadata all-gathered over the process group	ncclCommWindowRegister (collective; NCCL exchanges internally)
Map peers	for each peer: cuMemImportFromShareableHandle → cuMemAddressReserve → cuMemMap → cuMemSetAccess ⇒ buffer_ptrs[r]	ncclGetPeerDevicePointer ⇒ buffer_ptrs[r]
Multicast	rank 0: cuMulticastCreate → cuMemExportToShareableHandle all ranks: cuMemImportFromShareableHandle → cuMulticastAddDevice → cuMulticastBindMem → cuMemMap ⇒ multicast_ptr	ncclGetLsaMultimemDevicePointer ⇒ multicast_ptr
Free	multicast first: cuMemUnmap → cuMulticastUnbind → cuMemRelease own and peer mappings: cuMemUnmap → cuMemRelease	ncclCommWindowDeregister → ncclMemFree

GPU[k] Allocation Layout

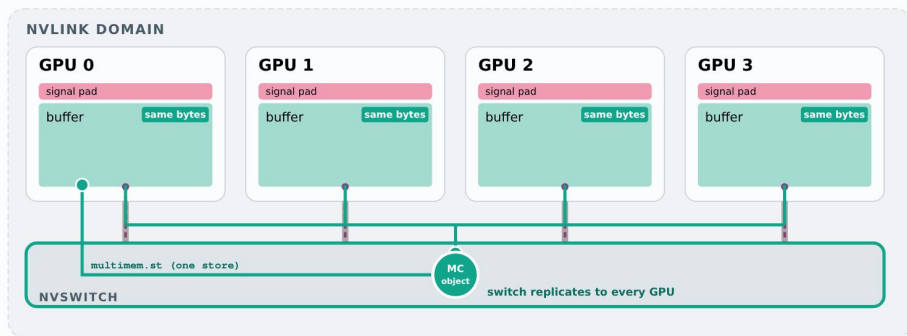
▼ pointer returned by `symm_mem.empty`



Signal pad: one 32-bit slot per (channel, peer) for barriers and put / wait signals

Multicast store

Rendezvous also binds every rank's buffer to one CUDA multicast object on the NVSwitch



- One `multimem.st` → the switch writes the bytes into every GPU
- The fan-out runs in the switch
- Needs multicast (Hopper+)

CUTE DSL · BROADCAST WITH ONE MULTIMEM.ST

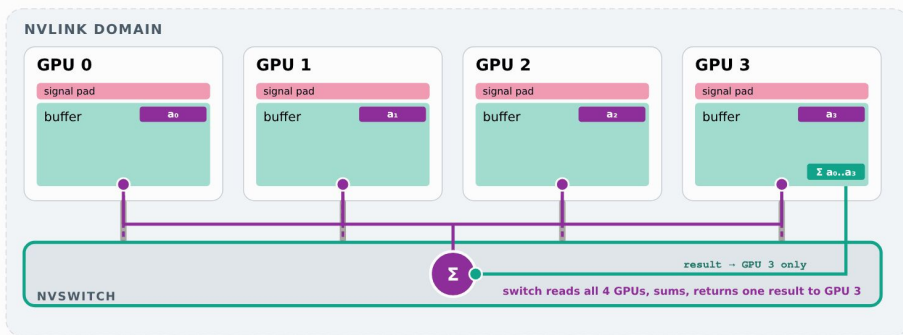
```
@cute.kernel
def broadcast(src: cute.Tensor, mc_ptr: cutlass.Int64):
    i = cute.arch.block_idx()[0] * 256 +
    cute.arch.thread_idx()[0]
    mc = cute.make_ptr(cutlass.Int32, mc_ptr,
    cute.AddressSpace.gmem)
    # one store to the multicast address:
    # the switch writes it into every GPU's buffer
    utils.distributed.multimem_st(mc + i, src[i].ir_value())

@cute.jit
def launch(src, mc_ptr: cutlass.Int64):
    broadcast(src, mc_ptr).launch(
        grid=[cute.size(src) // 256, 1, 1], block=[256, 1, 1])

launch(x.view(torch.int32), hdl.multicast_ptr)
```

In-switch reduction

Multicast reduction reads the same offset on every GPU; the NVSwitch does the reduction and returns one result.



- In-switch load + reduction using `multimem` instructions on NVSwitch
- In-switch reduction requires NVLink Sharp (NVLS)
- Example: `ld_reduce` my slice, then `multimem.st` it: two-shot all-reduce using multicast

CUTE DSL · TWO-SHOT ALL-REDUCE WITH `LD_REDUCE` + `ST`

```
@cute.kernel # each GPU reduces its 1/W slice, in place
def all_reduce_multimem(t: cute.Tensor, mc_ptr: cutlass.Int64,
                        rank: cutlass.Constexpr, W: cutlass.Constexpr):
    tid = cute.arch.block_idx()[0] * 256 + cute.arch.thread_idx()[0]
    i = rank * (cute.size(t) // W) + tid # my slice
    mc = cute.make_ptr(cutlass.Float32, mc_ptr, cute.AddressSpace.gmem)
    # one load: the switch returns the sum over all GPUs
    (s,) = utils.distributed.multimem_ld_reduce(
        mc + i, dtype=cutlass.Float32, num_elements=1)
    # one store: the sum lands on every GPU
    utils.distributed.multimem_st(mc + i, s)

@cute.jit
def launch(t, mc_ptr: cutlass.Int64,
           rank: cutlass.Constexpr, W: cutlass.Constexpr):
    all_reduce_multimem(t, mc_ptr, rank, W).launch(
        grid=[cute.size(t) // W // 256, 1, 1], block=[256, 1, 1])

launch(t, hdl.multicast_ptr, rank, world)
```

Composability with Kernel DSLs

Let's look at few examples of one-shot all-reduce – same tensors, same handle while using your favorite kernel DSL

HOST · PYTORCH

```
t = symm_mem.empty(n, dtype=torch.bfloat16, device=dev) # this rank's symmetric buffer
hdl = symm_mem.rendezvous(t, group)
t.copy_(x) # write this rank's data into its own buffer
hdl.barrier() # every rank's data is in place before peers read it
ptrs = torch.tensor(hdl.buffer_ptrs, dtype=torch.int64, device=dev) # Triton, CuTe DSL
peers = [hdl.get_buffer(r, t.shape, t.dtype) for r in range(hdl.world_size)] # Helion
out = torch.empty(n, dtype=torch.float32, device=dev)
```

TRITON

```
@triton.jit
def one_shot_all_reduce(ptrs, out, W: tl.constexpr,
                       BLOCK: tl.constexpr):
    offs = tl.program_id(0) * BLOCK + tl.arange(0,
BLOCK)
    acc = tl.zeros([BLOCK], tl.float32)
    for r in tl.static_range(W):
        peer = tl.load(ptrs + r).to(
            tl.pointer_type(tl.bfloat16))
        acc += tl.load(peer + offs) # GPU r
    tl.store(out + offs, acc)

one_shot_all_reduce[(n // 1024,)](
    ptrs, out, W=hdl.world_size, BLOCK=1024)
```

CUTE DSL

```
@cute.kernel
def one_shot_all_reduce(x: cute.Tensor,
                       ptrs: cute.Tensor, out: cute.Tensor,
                       W: cutlass.Constexpr):
    i = (cute.arch.block_idx()[0] * 256
        + cute.arch.thread_idx()[0])
    acc = cutlass.Float32(0.0)
    for r in cutlass.range_constexpr(W):
        p = cute.make_ptr(x.element_type, ptrs[r],
                           cute.AddressSpace.gmem)
        peer = cute.make_tensor(p, x.layout)
        acc += peer[i].to(cutlass.Float32) # GPU r
    out[i] = acc
```

HELION

```
@helion.jit
def one_shot_all_reduce(
    peers: tuple[torch.Tensor, ...],
    out: torch.Tensor) -> torch.Tensor:
    for tile in hl.tile(out.size(0)):
        acc = hl.zeros([tile], dtype=torch.float32)
        for p in peers: # one tensor per GPU
            acc += p[tile] # load from GPU r
    return out

one_shot_all_reduce(tuple(peers), out)
```



Community Adoption

Community is building on symmetric memory

Use cases enabled by symmetric memory are expanding and an ecosystem of projects has evolved over the last one year

OVERLAP AND FUSION

GEMM + collective

All-gather, reduce-scatter or all-reduce runs inside the GEMM, tile by tile.

Copy-engine transfer

The copy engine moves the data.
The GEMM keeps all SMs.

FAST INFERENCE

Low-latency collectives

One-shot, two-shot, multimem and Lamport all-reduce patterns

Collective + norm fusion

Latency-bound regime fused kernels

MEGAKERNELS

MoE in one kernel

Dispatch, the expert GEMMs and combine run in one launch without any all-to-all or host sync.

FASTER DIST.* COLLECTIVES

Symmetric Mempool

Allocate from the pool. NCCL selects its symmetric kernels. With the zero-CTA policy, all-gather and all-to-all run on the copy engine.

Use cases

Training frameworks, inference engines, and kernel libraries are building on PyTorch Symmetric Memory

PROJECT		OVERLAP AND FUSION		FAST INFERENCE		MEGAKERNELS	FASTER DIST.*
		GEMM + collective	Copy-engine transfer	Low-latency collectives	Collective + norm fusion	Fused MoE/EP kernels	Symmetric mempools
Training	TorchTitan	✓	✓			✓	✓
	Megatron-Core				✓	✓	✓
	OLMo-core	✓	✓				
	DeepSpeed		✓				
Inference	vLLM	✓		✓	✓	✓	
	SGLang	✓	✓	✓	✓	✓	
	TensorRT-LLM			✓		✓	
Kernel libraries	CUTLASS/CUTE DSL	✓					
	QuACK	✓	✓				
	FlashInfer	✓		✓		✓	
	TransformerEngine			✓		✓	
	DistMoE					✓	

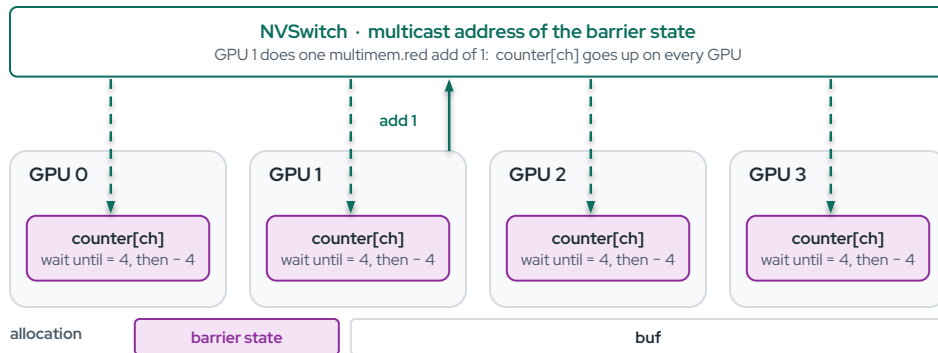
Multimem barrier

`hdl.barrier()`: each rank adds 1 to a counter on every GPU with one `multimem.red`, then waits on its own copy.

PSEUDO CODE · symmetric memory calls in purple

```
# once: rendezvous also maps the hidden barrier state
buf = symm_mem.empty(n, dtype=torch.bfloat16)
hdl = symm_mem.rendezvous(buf, group)
# CUDA backend + multicast: one tiny kernel
hdl.barrier(channel=0)

def multimem_barrier(local_state, mc_state, ch, world):
    # +1 on every GPU at once
    multimem_red_add(mc_state + ch, 1)
    while ld_acquire(local_state + ch) < world:
        pass # wait: all ranks arrived
    # reset own copy for reuse
    red_add(local_state + ch, -world)
```



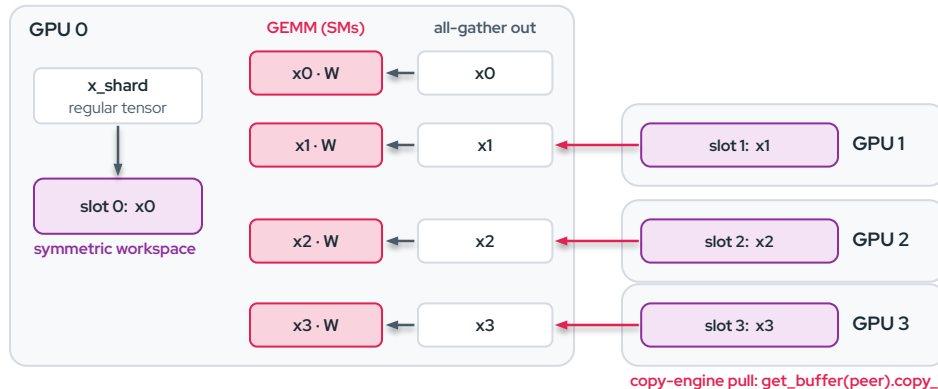
GEMM + collective

Example: all-gather hides behind the GEMM with symmetric memory workspace as staging area for peer access

PSEUDO CODE · symmetric memory calls in purple

```
# once per group, cached (get_symm_mem_workspace)
buf = symm_mem.empty(nbytes, dtype=torch.uint8)
hdl = symm_mem.rendezvous(buf, group)

def fused_all_gather_matmul(x_shard, w):
    hdl.barrier(channel=0)           # buffer is free
    hdl.get_buffer(rank, shape, dtype).copy_(x_shard) # publish
    hdl.barrier(channel=1)           # every slot is filled
    out[rank] = x_shard @ w
    for step in range(1, world):
        peer = (rank + step) % world
        # copy-engine pull from the peer's slot
        x[peer].copy_(hdl.get_buffer(peer, shape, dtype))
        out[peer] = x[peer] @ w      # GEMM on the landed chunk
    hdl.barrier(channel=0)           # buffer can be reused
    return x, out
```



FUSED all-gather + matmul, chunk by chunk. GEMM + reduce-scatter reuses the workspace; its final sum runs on SMs.
API `torch.ops.symm_mem.fused_all_gather_matmul(x_shard, [w], gather_dim=0, group_name=tp.group_name)`

Reduce-scatter + residual + RMSNorm + all-gather in one kernel

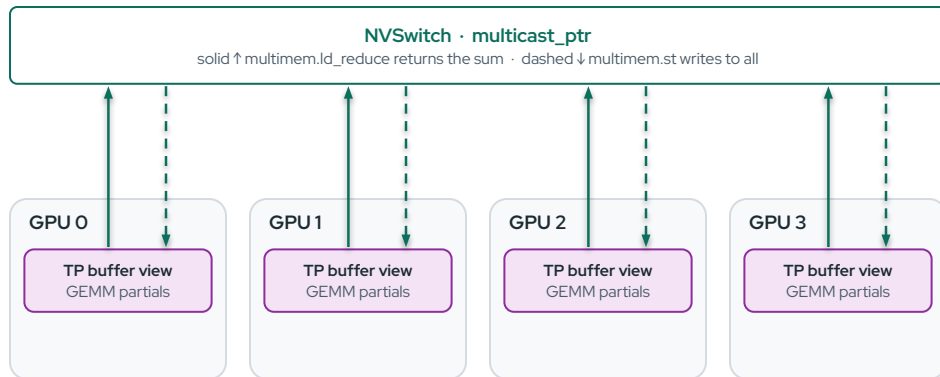
Example from megatron — fuse several collectives and norm in one kernel

PSEUDO CODE · symmetric memory calls in purple

```
# once per TP group
buf = symm_mem.empty(512 MB); hdl = symm_mem.rendezvous(buf, tp)

def row_parallel_linear(x, w, residual, norm_w):
    y = buf_view(tokens, hidden) # view of the TP buffer
    torch.matmul(x, w.t(), out=y) # partial sums, no copy
    fused_rs_add_norm_ag[grid](hdl.multicast_ptr,
                               hdl.signal_pad_ptrs_dev, residual, norm_w)
    return residual, y # y: normed output, every GPU

@triton.jit
def fused_rs_add_norm_ag(mc, pads, residual, norm_w):
    symm_mem_sync(pads) # all partials are written
    for t in my_tokens:
        s = multimem_ld_reduce(mc + t) + residual[t] # sum over GPUs
        residual[t] = s
        multimem_st(mc + t, rms_norm(s, norm_w)) # write to every GPU
    symm_mem_sync(pads) # outputs are visible
```



WHY SYMMETRIC MEMORY

- GEMM writes into a view of the TP buffer
- Reduction and fan-out run in the NVSwitch
- Output stays in the buffer: the next layer skips its norm + all-gather

FUSED reduce-scatter, residual add, RMSNorm and all-gather. Before: six steps. After: GEMM + one kernel.

API fused_multimem_rs_add_norm_ag(residual, buf, hdl, residual, norm_w, eps) # megatron/core/inference

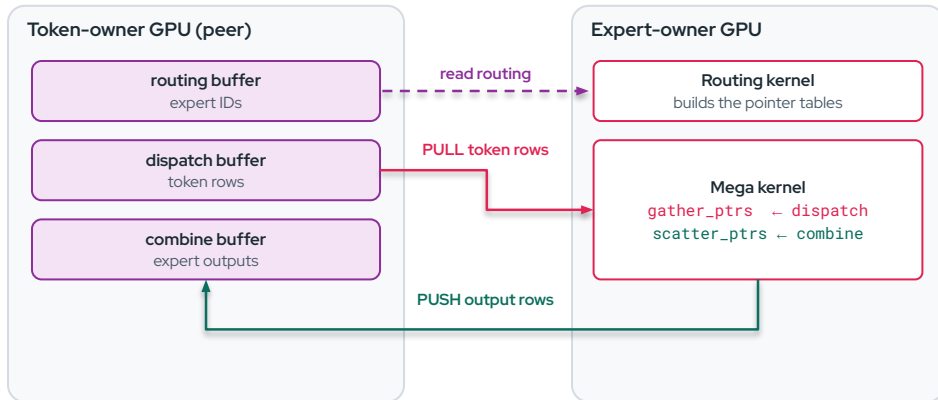
MoE in one launch: DistMoE

One kernel does dispatch, both expert GEMMs and combine. It reaches peers via symmetric memory `buffer_ptr`s

PSEUDO CODE · symmetric memory calls in purple

```
# once per context (create_context)
routing = symm_mem.empty(routing_bytes, dtype=torch.uint8)
dispatch = symm_mem.empty(token_bytes, dtype=torch.uint8)
combine = symm_mem.empty(token_bytes, dtype=torch.uint8)
flags = symm_mem.empty(n_channels, dtype=torch.int32)
hdl_routing, hdl_dispatch, hdl_combine, hdl_flags = [
    symm_mem.rendezvous(t, ep_group)
    for t in (routing, dispatch, combine, flags)]

def routed_experts(x, ids, scores, w13, w2):
    write(routing, ids); write(dispatch, x) # publish: local writes
    multimm_barrier(hdl_flags.multicast_ptr) # published data is visible
    gather_ptrs, scatter_ptrs = routing_kernel(
        hdl_routing.buffer_ptrs, # read peer routing
        hdl_dispatch.buffer_ptrs, hdl_combine.buffer_ptrs)
    mega_kernel(gather_ptrs, scatter_ptrs, w13, w2)
    # loads rows from peer dispatch, stores rows into peer combine
    multimm_barrier(hdl_flags.multicast_ptr) # outputs are visible
    return (read(combine) * scores).sum(top_k) # local read
```



WHY SYMMETRIC MEMORY

- Three buffers, created once per context
- The kernel gets plain int64 pointer tables: no collective call inside
- Barrier flags: one small symmetric buffer, updated through multicast

FUSED dispatch, W13, SwiGLU, W2 and combine in one chunk-pipelined launch.

API `ctx = dist_moe.create_context(...)` `dist_moe.routed_experts(x, ids, scores, w13, w2, ctx)`



Symmetric MemPools

Using regular collectives

Symmetric Mempools allow you to use your regular collectives api with symmetric memory. Couple of ways to do it..

A · REGISTER A POOL WITH THE PROCESS GROUP

```
backend = dist.get_backend_impl(pg, torch.device("cuda"))
pool = torch.cuda.MemPool(backend.mem_allocator) # ncclMemAlloc
with torch.cuda.use_mem_pool(pool):
    x = torch.empty(n, device="cuda")
backend.register_mem_pool(pool, symm=True)
dist.all_reduce(x) # → ncclSymkDevKernel_*
```

B · ALLOCATE WITH THE NCCL BACKEND

```
symm_mem.set_backend("NCCL")
x = symm_mem.empty(n, device="cuda:0")
hdl = symm_mem.rendezvous(x, group)
dist.all_reduce(x) # → ncclSymkDevKernel_*
my_kernel[grid](hdl.buffer_ptrs_dev, hdl.multicast_ptr, ...)
```

Symmetric kernels: all_gather, all_reduce, reduce_scatter are supported. Other collectives fall back to standard NCCL ring or tree collectives. As NCCL supports more symmetric kernels, should automatically be supported via torch.distributed apis

NCCL'S SYMMETRIC KERNELS (NCCL 2.32+) name: ncclSymkDevKernel_<Coll>_<Algo>_<red>_<type>

	LL low latency	LLMC low latency, multitem	LD / ST peer loads, stores	LDMC / STMC NVLS multitem	TMA bulk copies	GIN multi-node
AllGather any type	LL	LLMC	ST	STMC	TmaST · TmaSTMC	RailRing_LsaSTMC
AllReduce sum	AGxLL_R	AGxLLMC_R	RSxLD_AGxST	RSxLDMC_AGxSTMC	RSxTmaLD_AGxTmaST	—
ReduceScatter sum, avg	LL	—	LD	LDMC	TmaLD	RailA2A_LsaLD · LDMC

Using regular collectives

Keep your `dist.*` calls. When the tensor is in a registered symmetric window, NCCL selects its symmetric kernels automatically

EXAMPLE

```

symm_mem.set_backend("NCCL")
pool = symm_mem.get_mem_pool(device) # segments come from ncclMemAlloc

with torch.cuda.use_mem_pool(pool):
    out = torch.mm(a, w) # local output is in symmetric memory
    symm_mem.rendezvous(out, group) # out is in symmetric window

dist.all_reduce(out, group=group) # calls symmetric kernels

```

MEMPOOL REGISTRATION

WHEN	PYTORCH	NCCL
Once	<code>use_mem_pool</code> → pool segment	<code>ncclMemAlloc</code>
Once	<code>rendezvous</code> or <code>register_mem_pool</code>	<code>ncclCommWindowRegister</code> (<code>COLL_SYMMETRIC</code>)
Each step	<code>dist.all_reduce(out)</code> — unchanged	windows found → <code>ncclSymkDevKernel_*</code>



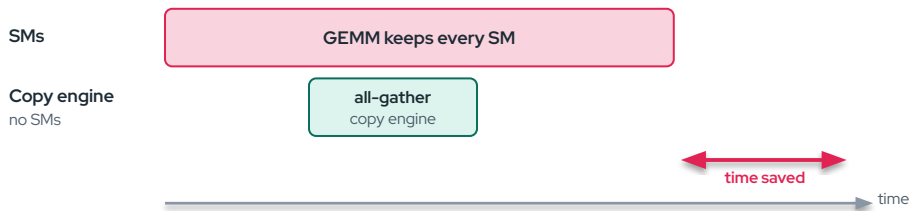
Copy-engine collectives: move data without SMs

Set NCCL's zero-CTA policy and allocate from the symmetric pool. All-gather and all-to-all then run on the copy engine.

DEFAULT · NCCL KERNEL ON SMs



ZERO-CTA POLICY · COPY ENGINE



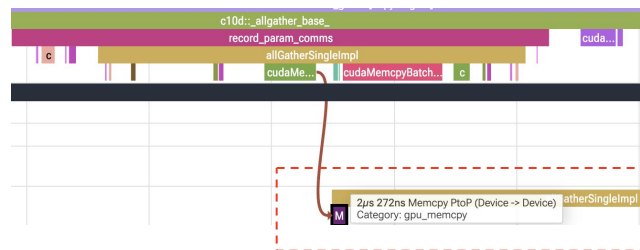
- NCCL runs pure data movement kernels such as all-gather, all-to-all on the copy engine
- Buffer must be in a symmetric window
- Needs NCCL 2.28+. From NCCL 2.30, all-gather and all-to-all also work across nodes

1 · ENABLE: ZERO-CTA POLICY ON THE PROCESS GROUP

```
opts = dist.ProcessGroupNCCL.Options()
opts.config.cta_policy = \
    dist.ProcessGroupNCCL.NCCL_CTA_POLICY_ZERO # or NCCL_CTA_POLICY=ZERO
dist.init_process_group("nccl", pg_options=opts, device_id=dev)
```

2 · ALLOCATE FROM THE SYMMETRIC POOL, KEEP DIST.*

```
symm_mem.set_backend("NCCL")
with torch.cuda.use_mem_pool(symm_mem.get_mem_pool(dev)):
    out = torch.empty(n * world, device=dev)
    symm_mem.rendevvous(out, group) # symmetric window
dist.all_gather_single(out, inp) # copy engine, zero SMs
```



CFT Transport + Symmetric Memory

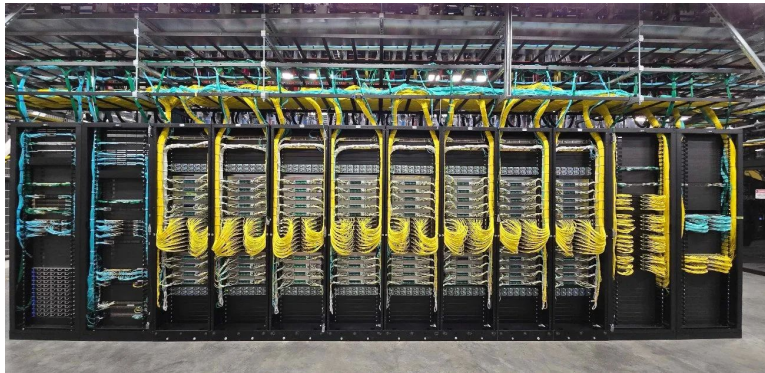
The background of the slide is an abstract composition of various organic and geometric shapes. It features several concentric circles and arcs in shades of red, pink, and orange. There are also darker, almost black, curved shapes that create a sense of depth and movement. The overall effect is a vibrant, layered, and somewhat chaotic visual field.

Why CFT transport?

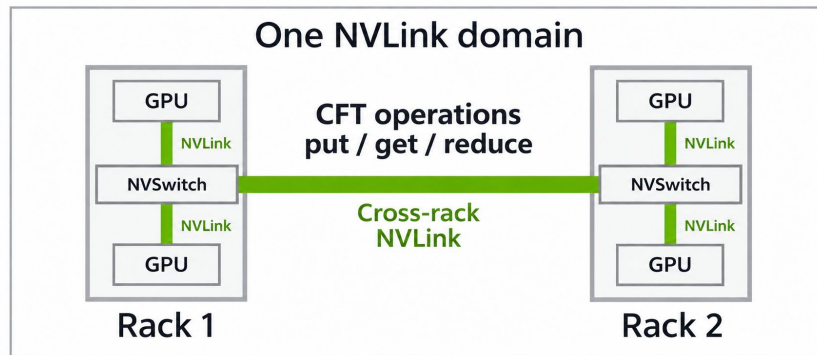
Communication primitives for larger NVLink fabrics

NVL576: 8 racks, 576 GPUs

Announced architecture. Polyphex prototype shown below.



NVLink also connects the racks



01 Cross-rack NVLink

Endpoint + offset addressing avoids mapping every peer's memory.

02 Counted writes

Data arrival advances a byte counter.
Fewer separate fence + flag steps.

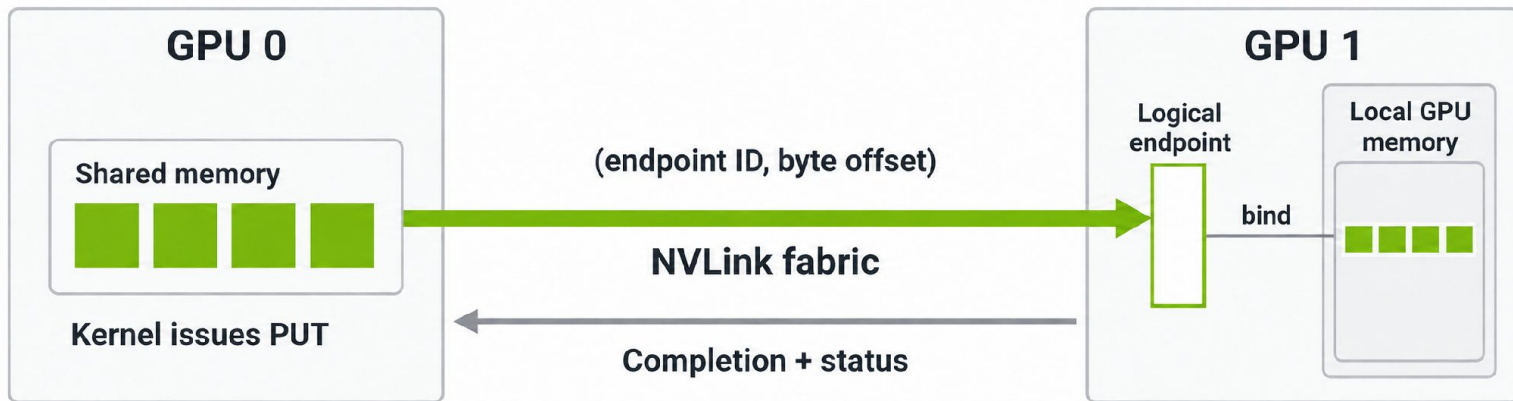
03 Error reporting

Completion reports fabric errors.
Software can choose recovery.

Sources: [NVIDIA NVL576 architecture](#) · [NCCL counted operations](#) · [CUDA error-status API](#)

CFT addresses remote memory by endpoint + offset

Compute Fabric Transport: asynchronous GPU communication over NVLink



Setup Create endpoints, bind buffers, exchange handles.

Why it matters No peer VA mapping. Explicit error status.

Sources: [CUDA Programming Guide §4.18](#)

CFT example: a kernel writes four values to GPU 1

GPU 0 sends [1, 2, 3, 4] to GPU 1's symmetric tensor x[256:260].

PYTORCH: SELECT GPU 1

```
id, base = (  
    hdl.get_peer_cft_handle(1)  
)  
offset = base + 256 * 4
```

Pass id and offset to the kernel.

GPU KERNEL: SEND 16 BYTES

```
// smem = [1, 2, 3, 4]  
cft.put(g, id, offset,  
        smem, 16);  
cft.submit(g);  
cft.flush(g);
```

PUT, submit, wait for completion.

GPU 1 reads [1, 2, 3, 4] after an ordered signal + wait.

Core excerpt. Buffer setup, shared-memory staging, fences, and error handling omitted.

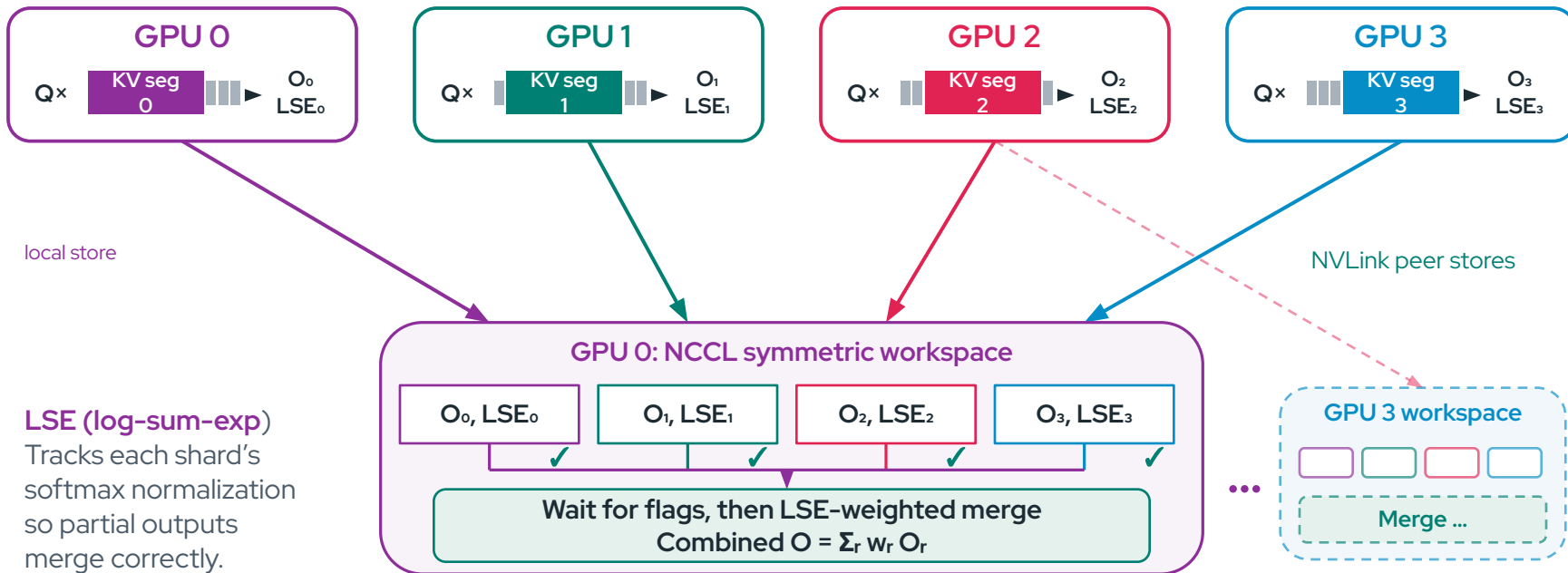
Sources: [PyTorch Symmetric Memory](#) · [NCCL CFT Device API](#)



Case Studies: FlashInfer

Decode context parallelism (DCP): fused attention combine

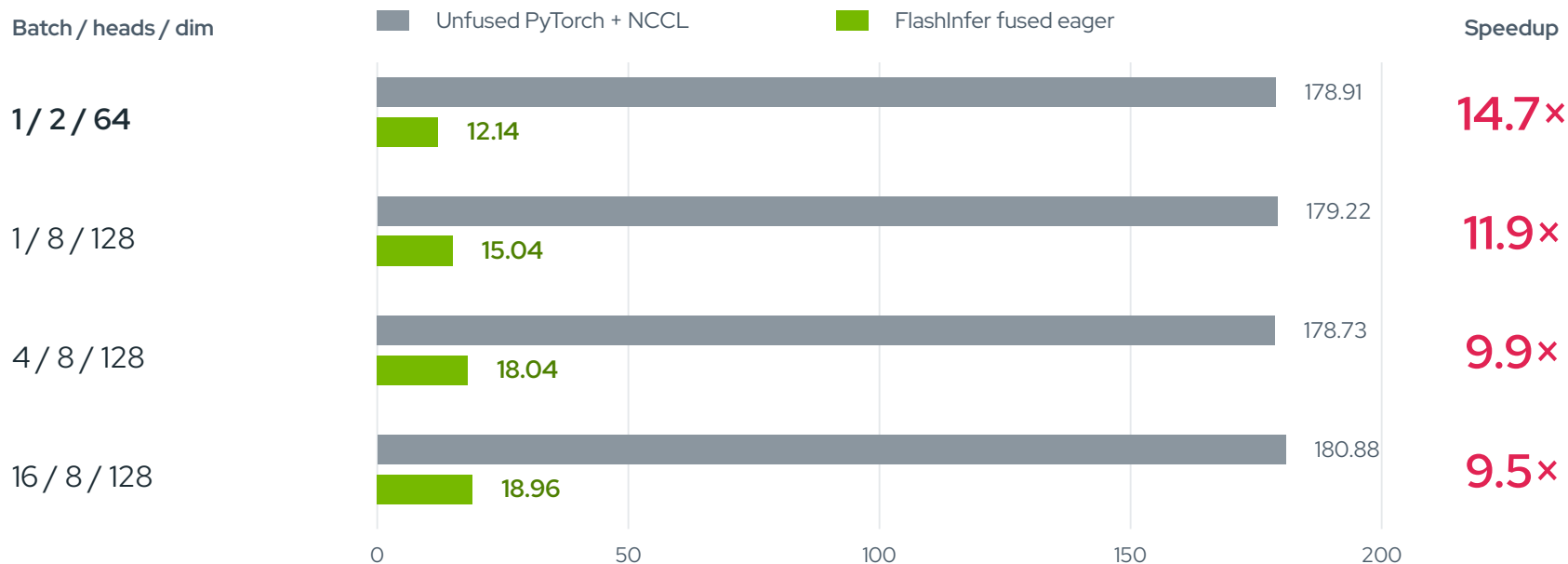
DCP splits the key/value cache by sequence. Each GPU produces a partial attention output O .



All-to-all: every GPU receives into its own workspace and merges its own slice.

Fused DCP combine cuts latency by 9.5–14.7×

4× H100 80 GB, BF16, CP=4. Operator latency in μ s, lower is better.



Max-rank median. Unfused = packing + two all-to-alls + PyTorch merge. [FlashInfer PR #4929](#)

Thank You!



Beyond Collectives: Building Fused Compute-Communication
Kernels with PyTorch Symmetric Memory

Acknowledgement

Thanks to all the contributors from

- Meta
- NVIDIA
- AMD
- Red Hat
- Intel
- Google,
- and many others

Learn more:

- [PyTorch Symmetric Memory Docs](#)
- [New programming paradigm in distributed AI](#)
- [PyTorch Symmetric Memory + NCCL Device APIs](#)
- [Advanced NCCL Tutorial: Beyond Collectives](#)