MSCCL++: Rethinking GPU Communication Abstractions for Cutting-Edge AI Applications

Presented by Yang Zhou
Thur 10/23

Abstract

Big problem:

- Modern cutting-edge AI applications are being developed over fast-evolving, heterogeneous, nascent hardware devices.
- This requires frequent reworking of the AI software stack to adopt bottom-up changes from new hardware, which takes time for general-purpose software libraries (NCCL)

MSCCL++ provides both portability and performance by essentially a reusable lib

- a primitive interface provides a minimal hardware abstraction
- higher-level portable interfaces and specialized implementations

Background

Various communication channels:

- CPU-GPU: PCIe, NVLink-C2C
- Intra-node: NVLink, xGMI,
- Inter-node: RDMA (IB, RoCE)

Key challenge: people want high communication performance, but this requires writing custom communication code, often from scratch and taking long time.

- ML workloads move fast

Motivation examples

The custom communication of TensorRT-LLM outperforms NCCL in a wide range of LLM scenarios, especially when the data size is relatively small, while TensorRTLLM still uses NCCL for larger data sizes.

What is 1-shot allreduce?

Key idea

Separate high-level abstractions and optimizations from primitive hardware abstractions

- Application developers to optimize and fine-tune by enabling precise control of the hardware
- Library developers to support new hardware features by only providing a shallow layer of abstraction over it

Basically, they are building a reusable communication library with an abstraction for various communication channels.

NCCL examples

Hard-coded GPU kernels

Send, recv not flexible enough

Inflexible synchronization

```
1  global ringRS(in, nelem, ring)
2    send = ring.sendbuff; recv = ring.recvbuff;
3    sz = ring.buffSz;
4    prim = Primitives<half>
5      (tid, ring.buffSz, ring.prev, ring.next);
6
7    for (off = 0; off < nelem; off += sz)
8      //step 0: push data to the next GPU
9      idx = off + ring.ranks[ring.ranks-1]*sz;
10     prims.copy(in+idx, send, sz);
11     prims.send();
12
13     //k-2 steps: reduce and copy to next GPU
14     for (j = 2; j < ring.ranks; ++j)
15       rankDest = ring.ranks[ring.ranks-j];
16       idx = off + rankDest * buffSz;
17       prims.recv();
18       prim.reduce(in+idx, recv, send, sz, "+");
19       prims.send();
20
21     //step k-1: write the result for this rank
22     idx = off + ring.rank * buffSz;
23     prims.recv();
24     prims.reduce(recv, in+idx, sz, "+");
```

Figure 1. Ring ReduceScatter (simplified) in NCCL.

MSCCL++ overview

High-level APIs

+

Low-level interfaces

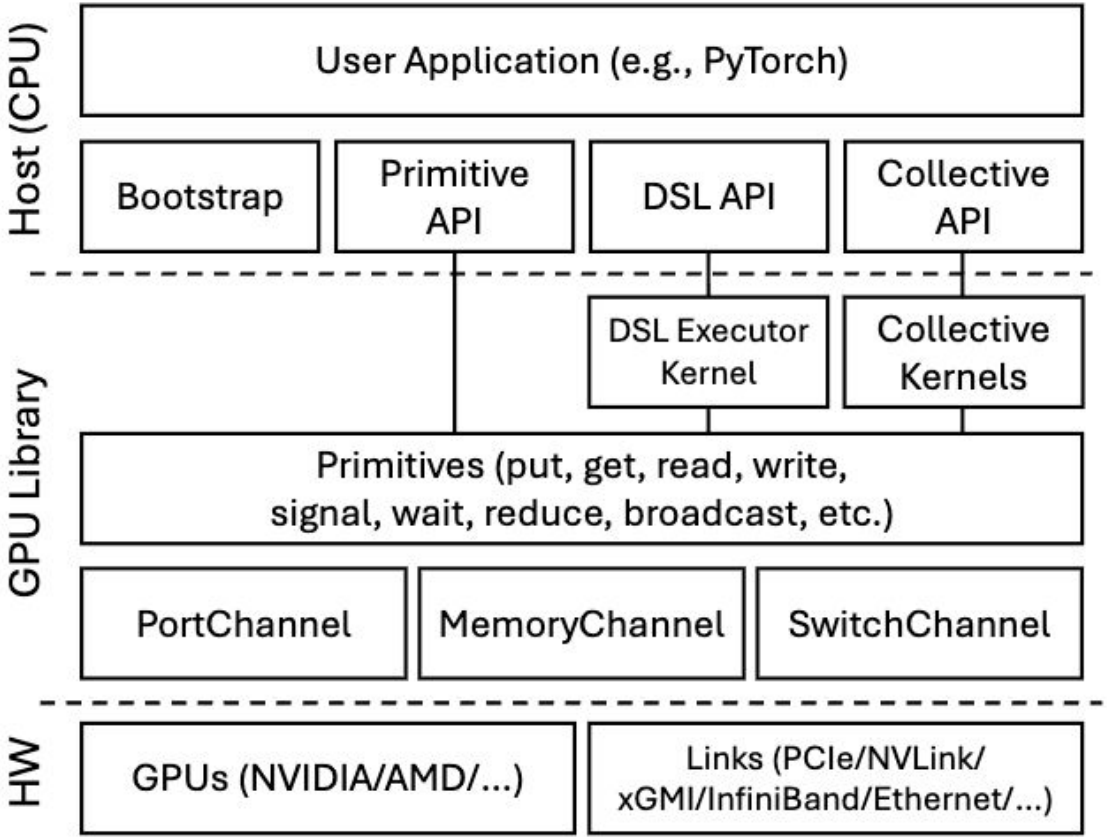

Figure 3. Overview of the MSCCL++ stack.

Communication channels

PortChannel: DMA engines on GPUs or RDMA NICs

MemoryChannel: peer-to-peer memory access over NVLink/xGMI

SwitchChannel: interconnection-switch-enabled multimem memory

Communication Primitives

All MSCCL++ primitives are defined as a method of a channel.

```
1 //async
2 put(src0, dst1, size)
3 //unsafe to reuse src0
4 signal() //async
5 flush() //sync
6 //safe to reuse src0
```

(a) GPU-0

```
//unsafe to read dst1
wait() //sync
//safe to read dst1
```

(b) GPU-1

Figure 4. MSCCL++ data transfer abstractions. `put` asynchronously transfers data from one GPU to another. `signal` and `wait` synchronize data transfer between GPUs. `flush` ensures the completion of preceded data transfer.

An example

Implementing a different ReduceScatter

```
1 global allPairsRS(count, gpus, channels[gpus])
2   sz = channel.scratchSz/gpus.num
3   count = count/gpus.num
4
5   for (off = 0; off < count; off += sz)
6     //Send 1/Nth data to each GPU
7     for (g = 0; g < gpus.num; g++)
8       idx = off + g * count;
9       channels[g].put(idx, sz*g, sz)
10      channels[g].signal()
11
12    //Reduce every pair of GPU
13    for (g = 0; g < gpus.num-1; g++)
14      channels[g].wait()
15      reduce(in + off, channels[g].scratch)
16
17    //barrier on all gpus
18    multiDeviceBarrier();
```

Figure 5. All-pairs ReduceScatter kernel in MSCCL++. Channels are initialized with source as input and destination as scratch buffer.

Implementation deepdive

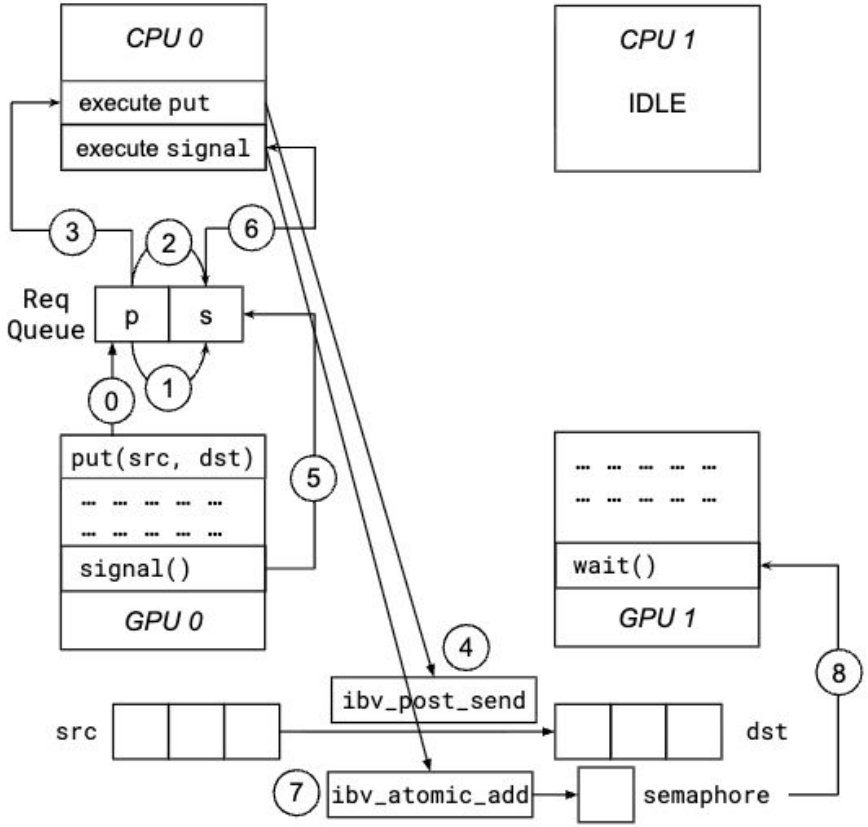

Figure 7. `PortChannel` workflow for IB from ⓪ when GPU 0 calls put primitive to ⑧ when GPU 1 receives the data.

Testbed

Env. Name	GPU	Intra-node Link	Network
A100-40G	NVIDIA A100 (40G) (8x/node)	NVLink 3.0	Mellanox HDR InfiniBand (200 Gb/s, 1x NIC/GPU)
A100-80G	NVIDIA A100 (80G) (8x/node)	NVLink 3.0	Mellanox HDR InfiniBand (200 Gb/s, 1x NIC/GPU)
H100	NVIDIA H100 (8x/node)	NVLink 4.0	Quantum-2 CX7 InfiniBand (400 Gb/s, 1x NIC/GPU)
MI300x	AMD MI300x (8x/node)	Infinity Fabric Gen 4	Quantum-2 CX7 InfiniBand (400 Gb/s, 1x NIC/GPU)

Table 1. List of environments used for evaluation.

Some results

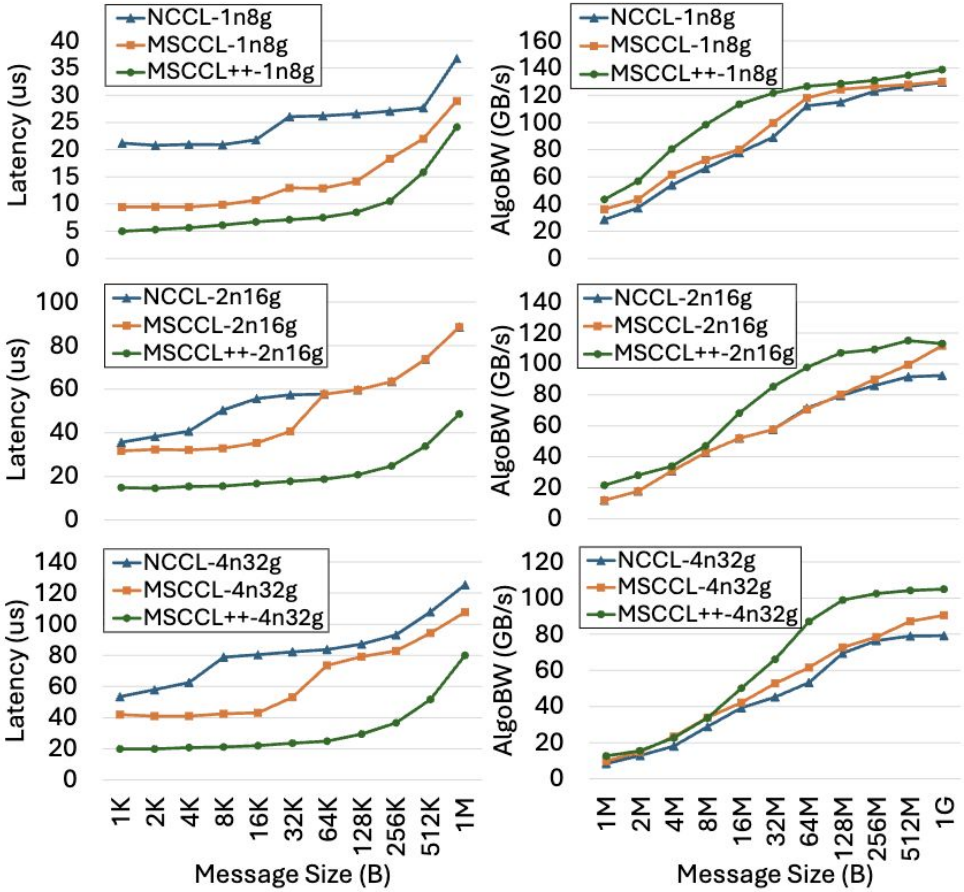

Figure 8. AllReduce, A100-40G. *M*n*N*g means *M* nodes and *N* GPUs. X-axis is message size (at the bottom of the figure).

Key takeaways

ML workloads is fast evolving, so do the system software (and so do hardware)

Communication is a cornerstone in ML workloads

We need a reusable library to develop custom communication strategies.

What so far we have covered

Datacenter networking:

- Microsecond and tail
- Topology design
- Congestion control
- How to use RDMA (efficiently)

Host networking:

- RDMA for GPU comm
- Efficient host TCP
- NCCL stack

What we will cover

LLM inference:

- Memory management
- Compute management
- Attention efficiency
- Holistic optimization

LLM training:

- Attention efficiency
- Training parallelism
- MoE (Mixture-of-Expert)
- Failure, auto-parallelism