CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

NCCL on Alps / GH200

User Lab Day 2025

Fabian Bösch

NVIDIA Collective Communication Library (NCCL)

- Efficient GPU-to-GPU collective communication
- Optimized for NVLink, PCIe, and network fabrics (Slingshot 11 on Alps/GH200)
- Widely used in ML frameworks (PyTorch, TensorFlow, JAX)
- Can be used for HPC applications, too!
- Complements MPI rather than replacing it
- AMD's version: RCCL

NCCL vs MPI

Feature	MPI	NCCL
Target	General-purpose comms	GPU-focused collectives
Collectives	Broad set (CPU & GPU)	Fewer, GPU-optimized
Point-to-point	Yes	Since v2.7
Bootstrapping	Built-in	Out-of-band (often via MPI)
Performance	Good (GPU-aware MPI exists)	Often faster for device buffers

Rule of thumb:

- **MPI for CPU ↔ CPU**
- **NCCL for GPU ↔ GPU**

How NCCL Works

- Collectives: `AllReduce`, `Broadcast`, `Reduce`, `AllGather`, `ReduceScatter`, `Send/Recv`
- Single-kernel implementation (comm + compute overlap)
- **Topology-aware** (NVLink, PCIe, network fabric)
- API is asynchronous and stream-based
  (operations are enqueued on CUDA streams)
- Requires a **communicator** for each GPU

NCCL API (C/C++ Example)

```
ncclResult_t ncclAllReduce(
    const void* sendbuff, void* recvbuff,
    size_t count, ncclDataType_t datatype, ncclRedOp_t op,
    ncclComm_t comm, cudaStream_t stream
);
```

- Similar to `MPI_Allreduce`, but:
  - `comm` is tied to one GPU
  - `stream` controls async execution
  - no tag matching
- **Asynchronous semantics**:
  operation runs in background on GPU, check via CUDA stream

NCCL in PyTorch

- PyTorch `DistributedDataParallel` (DDP) uses NCCL by default
- User API is simple:

```
import torch.distributed as dist

dist.init_process_group(
  backend="nccl",
  device_id=torch.device(f"cuda:{GPU_ID}"))
torch.cuda.set_device(GPU_ID)
```

- Framework handles communicators & collectives
- Critical operation: **gradient AllReduce** each training step
- Users just launch with `torchrun` or `srun`

GH200: Grace-Hopper C2C Link

- Cache-coherent interconnect between Grace CPU and Hopper GPU
- Two types of allocations:
  - `cudaMalloc` → HBM (fast, GPU-local, no CPU direct access)
  - `malloc/new` → host first-touch (HBM or DDR, cache-coherent, CPU+GPU)
- NCCL can transparently use either

What to Watch Out For

- **MPI + NCCL mixing**
  - Use MPI for host ↔️ host
  - Use NCCL for GPU ↔️ GPU
  - Disable GPU-aware MPI:
    `export MPICH_GPU_SUPPORT_ENABLED=0`
- **Deadlocks**
  - Can happen if mixing blocking MPI with NCCL collectives
- **Affinity**
  - CPU pinning & NUMA placement affect performance
- **Environment**
  - Several env vars need to be set for best performance
- **Debugging**
  - `export NCCL_DEBUG=INFO`

Using MPI (on host) concurrently with NCCL (on device)

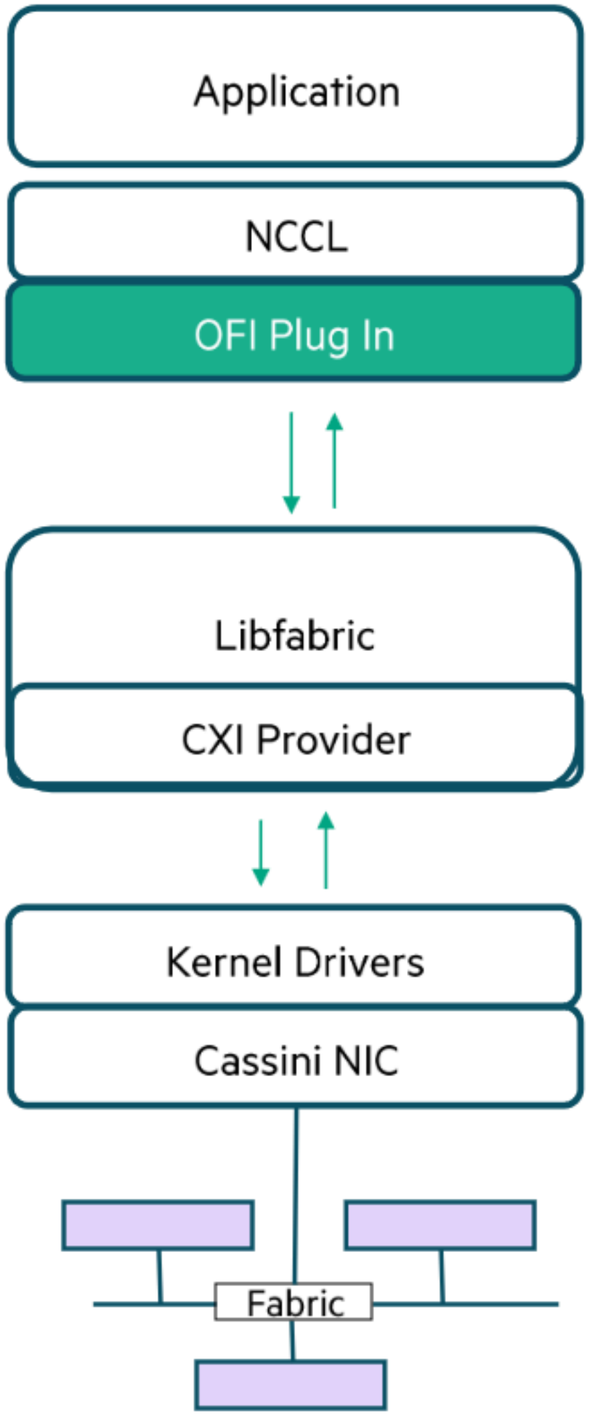
- MPI progress needs to be guaranteed
  - blocking calls such as `cudaStreamSynchronize` can lead to deadlocks
  - replace blocking calls with polling loops, and guarantee MPI progress

```
cudaError_t err = cudaErrorNotReady;
int flag;
while (err == cudaErrorNotReady) {
  err = cudaStreamQuery(stream);
  MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &flag, MPI_STATUS_IGNORE);
}
```

Alps Network Stack

HPE/Cray Slingshot Interconnect

- High-radix “Rosetta” switches
- ≤3 hops node-to-node
- Adaptive routing & congestion control
- Remote memory operations
- 25 GB/s per NIC (4 per node)



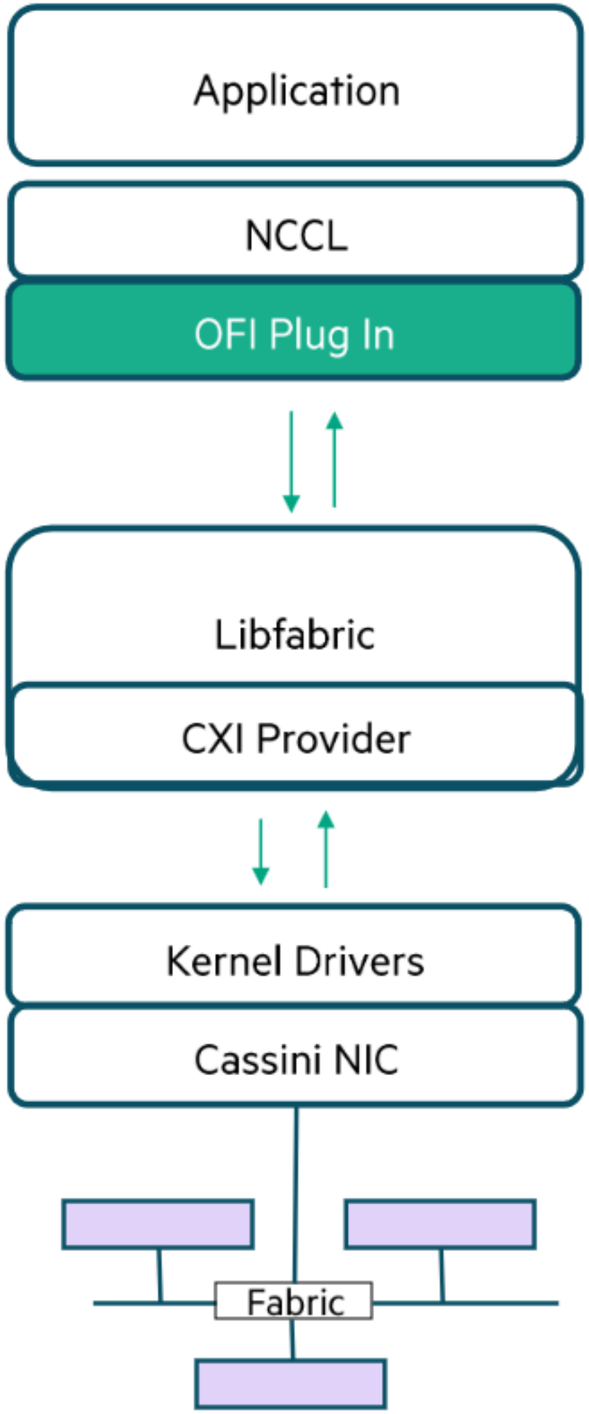
Network Stack (cont.)

Libfabric

- Open Fabrics Interfaces (OFI)
- Abstracts different network technologies
- Provider model

CXI Provider

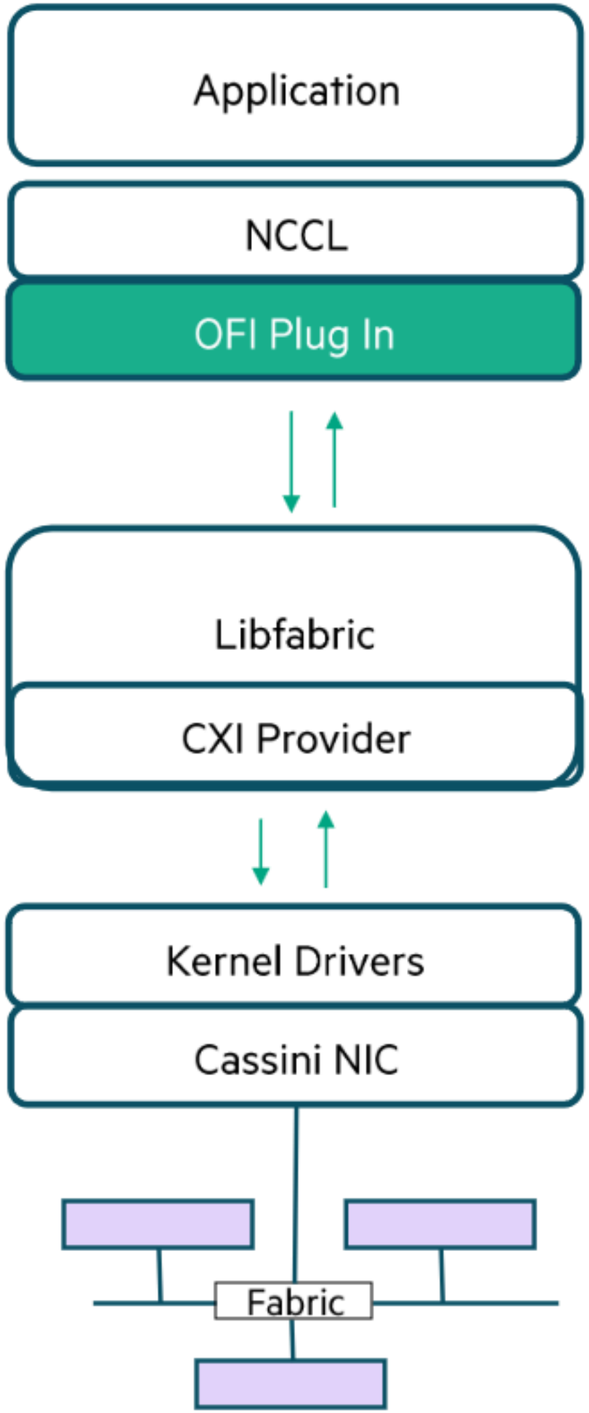
- Implements Slingshot support
- Message matching offloaded to NIC
- GPU-to-NIC RDMA
- Fabric-accelerated small reductions



aws-ofi-nccl Plugin

- Enables NCCL to use libfabric (CXI provider)
- Developed by NVIDIA + AWS, extended for HPE Slingshot
- Maps NCCL's transport APIs to libfabric
- Provides:
  - GPU RDMA over Slingshot
  - NIC offload for overlap of compute & comm
- Open source: https://github.com/aws/aws-ofi-nccl
- Plugin is available in uenvs & containers
- See CSCS docs for details

Warning: NCCL will fall back to TCP if the plugin is not available, resulting in poor performance!



Uenv and Container Support

Uenv

- require the aws-ofi-plugin
  - build with this spec: `aws-ofi-nccl@master`
- all CSCS-provided uenv's have the plugin
- environment variables need to be set in batch script

Containers with CE

- EDF file requires annotations to inject the aws-ofi-plugin
- environment variables can be set either in EDF or in sbatch script

```
[annotations]
com.hooks.aws_ofi_nccl.enabled = "true"
com.hooks.aws_ofi_nccl.variant = "cuda12"

[env]
NCCL_DEBUG=INFO
...
```

Required Environment (Slingshot)

```
export FI_MR_CACHE_MONITOR=userfaultfd
export FI_CXI_DISABLE_HOST_REGISTER=1
export FI_CXI_DEFAULT_CQ_SIZE=131072
export FI_CXI_DEFAULT_TX_SIZE=32768
export FI_CXI_RX_MATCH_MODE=software
export NCCL_CROSS_NIC=1
export NCCL_NET="AWS Libfabric"
export MPICH_GPU_SUPPORT_ENABLED=0
```

- `FI_MR_CACHE_MONITOR` : must not be `memhooks` (deadlocks)
- `FI_CXI_DISABLE_HOST_REGISTER` : avoids problematic host allocations
- `*_SIZE` : increase for large jobs
- `NCCL_CROSS_NIC` : improves scaling
- `NCCL_NET="AWS Libfabric"` : terminates if it fails to load the plugin
- Must disable GPU-aware MPI

Example: Training on Cluster

uenv:

```
#!/bin/bash
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=4

export TORCH_NCCL_ASYNC_ERROR_HANDLING=1
export MPICH_GPU_SUPPORT_ENABLED=0
export CUDA_CACHE_DISABLE=1
export TRITON_HOME=/dev/shm/
export NCCL_NET="AWS Libfabric"
export FI_CXI_DEFAULT_CQ_SIZE=131072
export FI_CXI_DEFAULT_TX_SIZE=32768
export FI_CXI_DISABLE_HOST_REGISTER=1
export FI_CXI_RX_MATCH_MODE=software
export FI_MR_CACHE_MONITOR=userfaultfd

export MASTER_ADDR=$(scontrol show hostnames $SLURM_JOB_NODELIST
    | head -n 1)
export MASTER_PORT=29500
export RANK=${SLURM_PROCID}

srun -ul --uenv=pytorch/v2.6.0 --view=default bash -c "
    LOCAL_RANK=\${SLURM_LOCALID} \
    WORLD_SIZE=\${SLURM_NTASKS} \
    python dist-train.py <dist-train-args>
"
```

container:

```
#!/bin/bash
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=4

export TORCH_NCCL_ASYNC_ERROR_HANDLING=1
export MPICH_GPU_SUPPORT_ENABLED=0
export CUDA_CACHE_DISABLE=1

export NCCL_NET="AWS Libfabric"

export MASTER_ADDR=$(scontrol show hostnames $SLURM_JOB_NODELIST
    | head -n 1)
export MASTER_PORT=29500
export RANK=${SLURM_PROCID}

srun -ul --environment=./ngc-pytorch-my-app-25.06.toml bash -c "
    LOCAL_RANK=\${SLURM_LOCALID} \
    WORLD_SIZE=\${SLURM_NTASKS} \
    python dist-train.py <dist-train-args>
"
```

Example: Training on Cluster: torchrun

uenv:

```
#!/bin/bash
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=1

export TORCH_NCCL_ASYNC_ERROR_HANDLING=1
export MPICH_GPU_SUPPORT_ENABLED=0
export CUDA_CACHE_DISABLE=1
export TRITON_HOME=/dev/shm/
export NCCL_NET="AWS Libfabric"
export FI_CXI_DEFAULT_CQ_SIZE=131072
export FI_CXI_DEFAULT_TX_SIZE=32768
export FI_CXI_DISABLE_HOST_REGISTER=1
export FI_CXI_RX_MATCH_MODE=software
export FI_MR_CACHE_MONITOR=userfaultfd

srun -ul --uenv=pytorch/v2.6.0 --view=default bash -c "
    python -m torch.distributed.run \
        --master-addr=\$(scontrol show hostnames
            \$SLURM_JOB_NODELIST | head -n 1) \
        --master-port=29500 \
        --nnodes=\${SLURM_NNODES} \
        --nproc-per-node=\${SLURM_GPUS_ON_NODE} \
        --node-rank=\${SLURM_PROCID} \
        dist-train.py <dist-train-args>
"
```

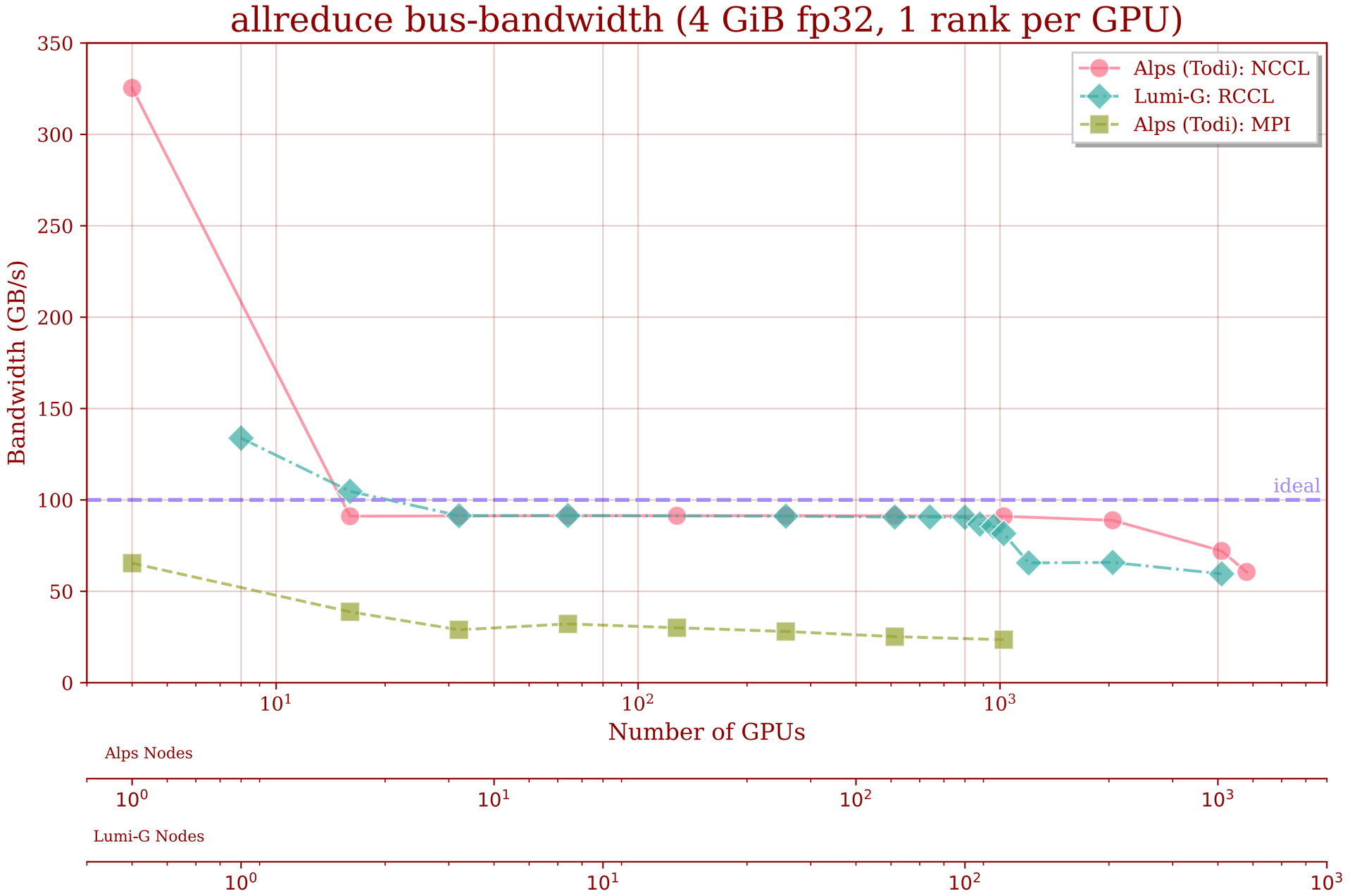
container:

```
#!/bin/bash
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=1

export TORCH_NCCL_ASYNC_ERROR_HANDLING=1
export MPICH_GPU_SUPPORT_ENABLED=0
export CUDA_CACHE_DISABLE=1

export NCCL_NET="AWS Libfabric"

srun -ul --environment=./ngc-pytorch-my-app-25.06.toml bash -c "
    python -m torch.distributed.run \
        --master-addr=\$(scontrol show hostnames
            \$SLURM_JOB_NODELIST | head -n 1) \
        --master-port=29500 \
        --nnodes=\${SLURM_NNODES} \
        --nproc-per-node=\${SLURM_GPUS_ON_NODE} \
        --node-rank=\${SLURM_PROCID} \
        dist-train.py <dist-train-args>
"
```


allreduce bus-bandwidth (4 GiB fp32, 1 rank per GPU)
Alps (Todi): NCCL
Lumi-G: RCCL
Alps (Todi): MPI
ideal
Bandwidth (GB/s)
0
50
100
150
200
250
300
350
10^1
10^2
10^3
Number of GPUs
Alps Nodes
10^0
10^1
10^2
10^3
Lumi-G Nodes
10^0
10^1
10^2
10^3

Resources

- NCCL User Guide: https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage.html
- CSCS docs: https://docs.cscs.ch/
- NCCL Tests: https://github.com/NVIDIA/nccl-tests
- PyTorch NCCL tests: https://github.com/stas00/ml-engineering/tree/master/network/benchmarks

Appendix: Advanced Topics

Asynchronous Semantics

- NCCL call returns
  - when the operation has been enqueued on the CUDA stream (*blocking* communicator)
    - may block the host thread before enqueueing (e.g. waiting for other ranks)
  - even before (*non-blocking* communicator)
    - requires waiting on communicator later on
- operation is executed asynchronously on devices
  - status can be checked using CUDA queries on the stream (e.g. using events)
  - `cudaStreamSynchronize` ensures completion of the collective

Group Calls: Avoiding Deadlocks

- single thread manages multiple devices
  - each device has its own communicator (and stream)
- sequential operations can deadlock (blocking semantics)

```
for (int i=0; i<nLocalDevs; i++) {
    ncclAllReduce(..., comm[i], stream[i]);
}
```

- solution: use group calls

```
ncclGroupStart();
for (int i=0; i<nLocalDevs; i++) {
    ncclAllReduce(..., comm[i], stream[i]);
}
ncclGroupEnd();
```

- effectively makes NCCL functions non-blocking (with a *blocking* communicator)
  - NCCL kernels may not be enqueued on the stream after function returns
  - synchronizing the stream can only be done after `ncclGroupEnd()`

Group Calls: Aggregating Operations

- aggregate communication operations -> single NCCL kernel launch: reduced overhead
- requires version 2.2 and later
- simply enclose operations by group calls

```
ncclGroupStart();
ncclBroadcast(sendbuff1, recvbuff1, count1, datatype, root, comm, stream);
ncclAllReduce(sendbuff2, recvbuff2, count2, datatype, comm, stream);
ncclAllReduce(sendbuff3, recvbuff3, count3, datatype, comm, stream);
ncclGroupEnd();
```

- multiple streams are allowed
  - will enforce a stream dependency
  - blocks all streams until the NCCL kernel completes

```
ncclGroupStart();
ncclBroadcast(sbuff1, rbuff1, count1, dtype, root, comm, stream1);
ncclAllReduce(sbuff2, rbuff2, count2, dtype, comm, stream2);
ncclAllReduce(sbuff3, rbuff3, count3, dtype, comm, stream3);
ncclGroupEnd();
```

- aggregation with multiple comms is allowed
  - may use nested group calls (optional)

```
ncclGroupStart();
for (int i=0; i<nlayers; i++) {
    ncclGroupStart();
    for (int g=0; g<ngpus; g++) {
        ncclAllReduce(sbuffs[g]+offsets[i], rbuffs[g]+offsets[i],
            counts[i], datatype[i], comms[g], streams[g]);
    }
    ncclGroupEnd();
}
ncclGroupEnd();
```

Group Calls: Point-to-Point Communications

- Point-to-Point requires version 2.7 and later

```
ncclResult_t ncclSend(const void* sendbuff, size_t count, ncclDataType_t datatype, int peer, ncclComm_t comm, cudaStream_t stream)
ncclResult_t ncclRecv(void* recvbuff, size_t count, ncclDataType_t datatype, int peer, ncclComm_t comm, cudaStream_t stream)
```

- use group calls to fuse multiple sends/recvs
  - form more complex communication patterns
  - scatter, gather, all-to-all, halo-exchange

```
ncclGroupStart();
if (rank == root) {
    for (int r=0; r<nranks; r++)
        ncclSend(sendbuff[r], size, type, r, comm, stream);
}
ncclRecv(recvbuff, size, type, root, comm, stream);
ncclGroupEnd();
```

Group Calls: non-blocking Communicators

- non-blocking communicators can also be used in group calls
- group functions become asynchronous
- requires synchronization of communicators

```
ncclGroupStart();
for (int g=0; g<ngpus; g++) {
    ncclAllReduce(sendbuffs[g]+offsets[i], recvbuffs[g]+offsets[i], counts[i], datatype[i], comms[g], streams[g]);
}
auto ret = ncclGroupEnd();
if (ret == ncclInProgress) {
    for (int g=0; g<ngpus; g++) {
        do {
            ncclCommGetAsyncError(comms[g], &ret);
        } while (ret == ncclInProgress);
        if (ret != ncclSuccess) break;
    }
}
if (ret != ncclSuccess) {
    // error
}

for (int g=0; g<ngpus; g++) {
    cudaStreamSynchronize(streams[g]);
}
```

Communicators and Communication Groups

- communication primitives transfer data among members of a communication `group`
- each group member corresponds to a CUDA device index
- a `communicator` is part of a particular `group`
- creating a communicator is a collective call

```
ncclResult_t ncclCommInitRank(ncclComm_t* comm, int nranks, ncclUniqueId commId, int rank)
ncclResult_t ncclCommInitRankConfig(ncclComm_t* comm, int nranks, ncclUniqueId commId, int rank, ncclConfig_t* config)
ncclResult_t ncclCommInitAll(ncclComm_t* comms, int ndev, const int* devlist)
ncclResult_t ncclCommSplit(ncclComm_t comm, int color, int key, ncclComm_t* newcomm, ncclConfig_t* config)
```

```
ncclComm_t comm;
auto state = ncclCommInitRank(&comm,
    world_size,
    unique_id,
    world_rank
);
```

- `world_size` : number of communicators in the group, typically number of GPUs in allocation
- `world_rank` : rank of this communicator, may not be equal to the MPI rank!

Unique IDs and Out-of-Band Initialization

- one unique id per communication group
  - must be equal for all ranks in the group
- disjoint groups require different unique ids

```
ncclResult_t ncclGetUniqueId(ncclUniqueId* uniqueId)
```

- unique id must be communicated (broadcast) out-of-band (e.g. using MPI)

```
ncclUniqueId id;
if (mpi_rank == 0) ncclGetUniqueId(&id);
MPI_Bcast(&id, sizeof(id), MPI_BYTE, 0, MPI_COMM_WORLD);
```

- multiple communicators for a single device
  - requires multiple unique ids

Blocking vs Non-Blocking Communicators

- Communicators are blocking by default
  - CPU host thread blocks until operation is enqueued on CUDA stream
  - may require group calls to avoid deadlocks (multiple comms per thread)
- Non-blocking Communicators
  - avoid these deadlocks
  - group calls are still required for aggregation
  - require manual synchronization
  - check `ncclCommGetAsyncError(comms, &state)` until `state != ncclInProgress`

```
ncclConfig_t config = NCCL_CONFIG_INITIALIZER;
config.blocking = 0;

auto state = ncclCommInitRankConfig(&comm, world_size, unique_id, world_rank, &config);
if (state != ncclSuccess && state != ncclInProgress) { /* error */ }
else {
    while(state == ncclInProgress) {
        check_nccl(ncclCommGetAsyncError(m_->comm, &state));
    }
    if (state != ncclSuccess) { /* error */ }
}
```

Host Concurrency

- NCCL primitives are
  - not thread-safe
  - reentrant -> multiple threads should use separate communicator objects

Device Concurrency

- multiple communicators for a single GPU
  - may work using different streams, but not guaranteed
  - may work if operations fit into GPU
  - may deadlock if NCCL collectives issue a CUDA operation which causes a device synchronization
  - safe: serialize access to device, use same order of operations on all ranks