NCCL: Multi-GPU Collective Communication

Yuke Wang
Rice University

February 27, 2026

Why GPU-Specific Collectives?

- Traditional MPI collectives are CPU-oriented
- GPUs have high bandwidth but asymmetric interconnects (PCIe, NVLink, NVSwitch)
- NCCL exploits topology and CUDA streams
- Enables overlap of communication & computation

Design Principles of NCCL

- Topology-aware rings and trees
- Bandwidth-optimal for large tensors
- Latency-optimal for small tensors
- Asynchronous stream integration
- Scalable intra-node and multi-node

Ring vs Tree AllReduce

Ring AllReduce:

- Each GPU sends/receives chunks in a ring
- Bandwidth-optimal for large messages

Tree AllReduce:

- Hierarchical reduction and broadcast
- Logarithmic latency
- Better for small messages

Full NCCL Multi-GPU Example (Headers & Macros)

```c
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <nccl.h>

#define N 1024

#define CHECK_CUDA(cmd) do { \
  cudaError_t e = cmd; \
  if (e != cudaSuccess) { \
    printf("CUDA error %s:%d '%s'\n", \
           __FILE__, __LINE__, cudaGetErrorString(e)); \
    exit(EXIT_FAILURE); \
  } \
} while(0)

#define CHECK_NCCL(cmd) do { \
  ncclResult_t r = cmd; \
  if (r != ncclSuccess) { \
    printf("NCCL error %s:%d '%s'\n", \
           __FILE__, __LINE__, ncclGetErrorString(r)); \
    exit(EXIT_FAILURE); \
  } \
} while(0)
```

Full Example: Allocation & Initialization

```
int main() {
    int nDev = 2;
    int devs[2] = {0,1};

    ncclComm_t comms[2];
    cudaStream_t streams[2];
    float *sendbuff[2], *recvbuff[2];

    // Allocate memory and create streams
    for (int i = 0; i < nDev; i++) {
        CHECK_CUDA(cudaSetDevice(devs[i]));
        CHECK_CUDA(cudaMalloc(&sendbuff[i], N*sizeof(float)));
        CHECK_CUDA(cudaMalloc(&recvbuff[i], N*sizeof(float)));
        CHECK_CUDA(cudaStreamCreate(&streams[i]));

        // Initialize data
        float host[N];
        for(int j=0;j<N;j++) host[j]=1.0f;
        CHECK_CUDA(cudaMemcpy(sendbuff[i], host,
                              N*sizeof(float),
                              cudaMemcpyHostToDevice));
    }

    // Initialize NCCL communicator
```

Full Example: Grouped AllReduce

```
// Grouped AllReduce
CHECK_NCCL(ncclGroupStart());
for(int i=0;i<nDev;i++){
    CHECK_CUDA(cudaSetDevice(devs[i]));
    CHECK_NCCL(ncclAllReduce(
        sendbuff[i],
        recvbuff[i],
        N,
        ncclFloat,
        ncclSum,
        comms[i],
        streams[i]));
}
CHECK_NCCL(ncclGroupEnd());

// Synchronize streams
for(int i=0;i<nDev;i++){
    CHECK_CUDA(cudaSetDevice(devs[i]));
    CHECK_CUDA(cudaStreamSynchronize(streams[i]));
}
```

Full Example: Verification & Cleanup

```
    // Copy results back and verify
    float result[N];
    CHECK_CUDA(cudaMemcpy(result, recvbuff[0],
                          N*sizeof(float),
                          cudaMemcpyDeviceToHost));

    printf("Expected value: %d\n", nDev);
    printf("First element: %f\n", result[0]);

    // Cleanup
    for(int i=0;i<nDev;i++){
        CHECK_CUDA(cudaFree(sendbuff[i]));
        CHECK_CUDA(cudaFree(recvbuff[i]));
        CHECK_NCCL(ncclCommDestroy(comms[i]));
    }

    printf("AllReduce completed successfully.\n");
    return 0;
}
```

Compile and Run

```
nvcc -o allreduce allreduce.cu -lnccl
./allreduce
```

Key Technical Insights

- **Ring Algorithm:** Bandwidth-optimal for large tensors
- **Tree Algorithm:** Low latency for small tensors
- **Topology-aware:** PCIe/NVLink/NVSwitch detection
- **Stream Integration:** Overlap compute and communication
- **Grouped Collectives:** Lower overhead & deadlock-free

Applications in Deep Learning

- AllReduce gradients efficiently across multiple GPUs
- Enables near-linear scaling
- Core of distributed PyTorch / TensorFlow training
- Minimizes communication bottlenecks

Scaling Intuition

- Small tensors: latency-bound $\rightarrow$ Tree AllReduce
- Large tensors: bandwidth-bound $\rightarrow$ Ring AllReduce
- Multi-node: combine intra-node rings + inter-node IB communication

Questions?