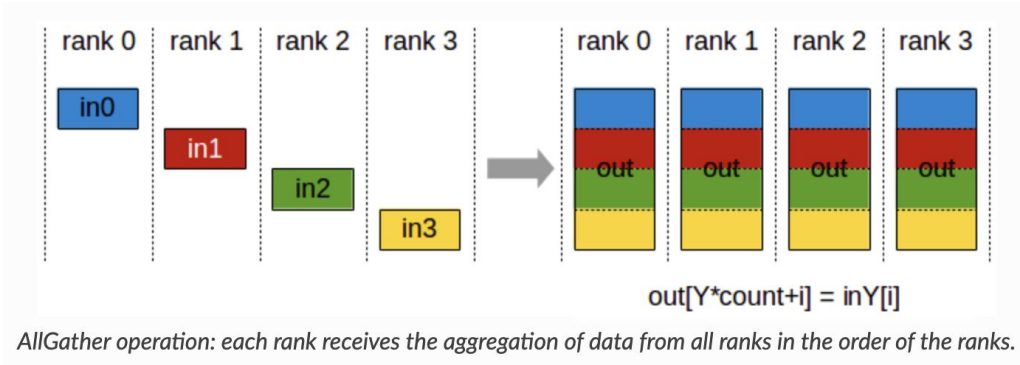
GPU Collective Communication (NCCL)

Dan Johnson

NCCL Overview

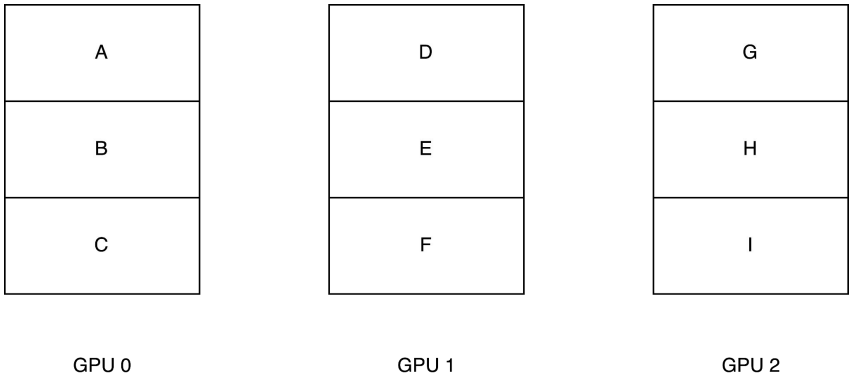
- NVIDIA Collective Communications Library
- Provides way for gpus to communicate data quickly
- Point-to-point and collective communication for GPUs
  - Scatter
  - Gather
  - All-to-all
  - AllReduce
  - Broadcast
  - Reduce
  - AllGather
  - ReduceScatter



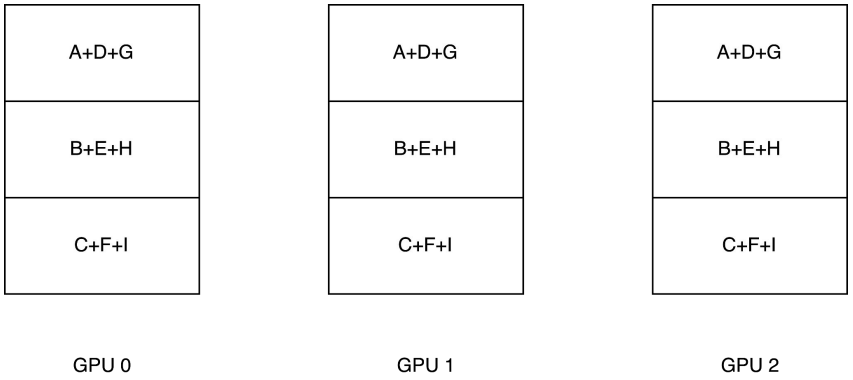
AllGather operation: each rank receives the aggregation of data from all ranks in the order of the ranks.

Image source: https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/collectives.html

AllReduce



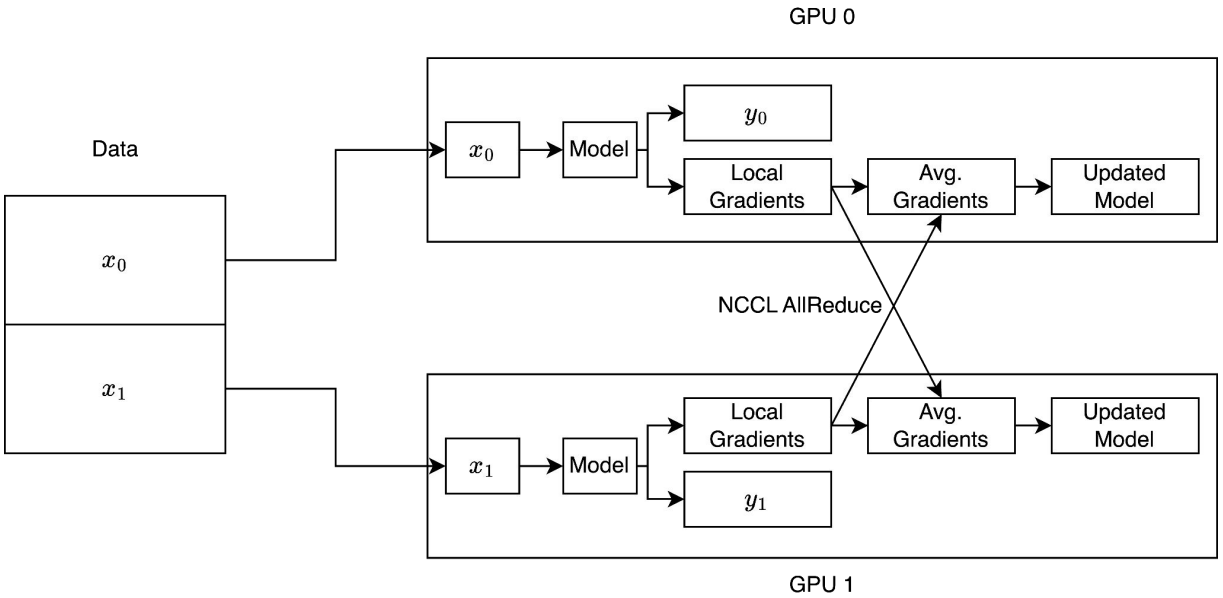
Before



After

Why do we need NCCL?

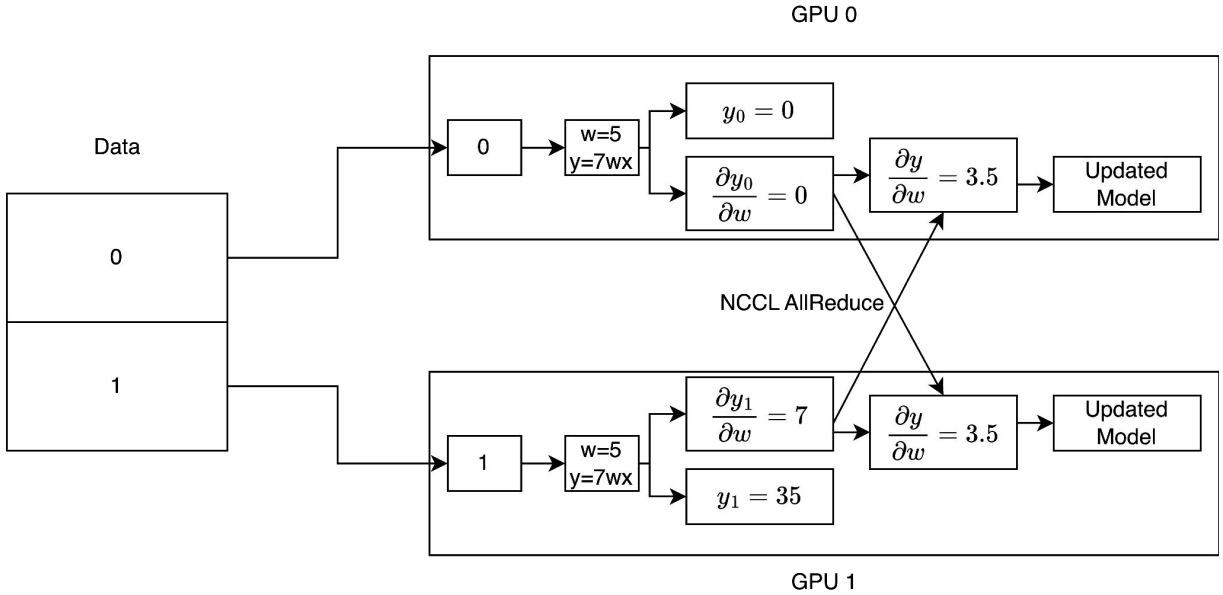
- Distributed data parallel (DDP) model example



Why do we need NCCL?

- Distributed data parallel (DDP) model example
  - Simple model: **y = w * 7 * x**
    - **w** is a tunable parameter, should be same across all ranks
    - **x** is the input to the model, different across ranks
  - Demo: ddp_simple.py

```
rank 0: y=w*7*x: 0.0=5.0*7*0.0
rank 0: dy/dw=7*x: 0.0
rank 1: y=w*7*x: 35.0=5.0*7*1.0
rank 1: dy/dw=7*x: 7.0
rank 0: reduced dy/dw: 3.5
rank 1: reduced dy/dw: 3.5
```



How does PyTorch use NCCL?

- Training example
- Let’s look at our example trace
- Somehow, this code is calling NCCL AllReduce
- Aside: CUDA streams
  - What is a CUDA stream?
  - Why are the AllReduces on their own stream?
- Good blog post with more detail
  - https://siboehm.com/articles/22/data-parallel-training

PyTorch DDP - Under the Hood

- “DDP uses autograd hooks registered at construction time to trigger gradients synchronizations… the Reducer kicks off an asynchronous allreduce on that bucket to calculate mean of gradients across all processes… When this is done, averaged gradients are written to the param.grad field of all parameters. So after the backward pass, the grad field on the same corresponding parameter across different DDP processes should be the same.”
  - https://pytorch.org/docs/master/notes/ddp.html
- Different communication backends including NCCL, MPI, Gloo can be used
- NCCL API is called by Reducer in ProcessGroupNCCL.cpp

ncclAllReduce API

ncclAllReduce

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

Reduce data arrays of length `count` in `sendbuff` using `op` operation and leaves identical copies of the result on each `recvbuff`.

In-place operation will happen if `sendbuff == recvbuff`.

Communicator Objects

- 1 GPU per CPU process
  - Root process generates uniqueId
  - Broadcast id to all processes (e.g. use MPI)
  - All processes initialize communicator with same id, unique rank
  - Each process then launches AllReduce Kernel
- Multiple GPUs on 1 CPU process
  - Generate uniqueId (no need to broadcast)
  - Loop through initializing each rank
  - Wrapper that does both for you (ncclCommInitAll)
  - Process launches all AllReduce Kernels

```
int myRank, nRanks;
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
MPI_Comm_size(MPI_COMM_WORLD, &nRanks);
```

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

```
ncclComm_t comm;
ncclCommInitRank(&comm, nRanks, id, myRank);
```

```
ncclResult_t ncclCommInitAll(ncclComm_t* comm, int ndev, const int* devlist) {
  ncclUniqueId Id;
  ncclGetUniqueId(&Id);
  ncclGroupStart();
  for (int i=0; i<ndev; i++) {
    cudaSetDevice(devlist[i]);
    ncclCommInitRank(comm+i, ndev, Id, i);
  }
  ncclGroupEnd();
}
```

Example NCCL AllReduce - Single Process

```
#include <stdio.h>
#include "cuda_runtime.h"
#include "nccl.h"

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


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

Source: https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/examples.html

Example NCCL AllReduce - Single Process

```
int main(int argc, char* argv[])
{
  ncclComm_t comms[4];


  //managing 4 devices
  int nDev = 4;
  int size = 32*1024*1024;
  int devs[4] = { 0, 1, 2, 3 };


  //allocating and initializing device buffers
  float** sendbuff = (float**)malloc(nDev * sizeof(float*));
  float** recvbuff = (float**)malloc(nDev * sizeof(float*));
  cudaStream_t* s = (cudaStream_t*)malloc(sizeof(cudaStream_t)*nDev);


  for (int i = 0; i < nDev; ++i) {
    CUDACHECK(cudaSetDevice(i));
    CUDACHECK(cudaMalloc(sendbuff + i, size * sizeof(float)));
    CUDACHECK(cudaMalloc(recvbuff + i, size * sizeof(float)));
    CUDACHECK(cudaMemset(sendbuff[i], 1, size * sizeof(float)));
    CUDACHECK(cudaMemset(recvbuff[i], 0, size * sizeof(float)));
    CUDACHECK(cudaStreamCreate(s+i));
  }
```

Source: https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/examples.html

Example NCCL AllReduce - Single Process

```
//initializing NCCL
NCCLCHECK(ncclCommInitAll(comms, nDev, devs));


 //calling NCCL communication API. Group API is required when using
 //multiple devices per thread
NCCLCHECK(ncclGroupStart());
for (int i = 0; i < nDev; ++i)
  NCCLCHECK(ncclAllReduce((const void*)sendbuff[i], (void*)recvbuff[i], size, ncclFloat, ncclSum
      comms[i], s[i]));
NCCLCHECK(ncclGroupEnd());


//synchronizing on CUDA streams to wait for completion of NCCL operation
for (int i = 0; i < nDev; ++i) {
  CUDACHECK(cudaSetDevice(i));
  CUDACHECK(cudaStreamSynchronize(s[i]));
}
```

Source: https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/examples.html

Example NCCL AllReduce - Single Process

```
  //free device buffers
  for (int i = 0; i < nDev; ++i) {
    CUDACHECK(cudaSetDevice(i));
    CUDACHECK(cudaFree(sendbuff[i]));
    CUDACHECK(cudaFree(recvbuff[i]));
  }


  //finalizing NCCL
  for(int i = 0; i < nDev; ++i)
      ncclCommDestroy(comms[i]);


  printf("Success \n");
  return 0;
}
```

Note: I think they forget to free the memory on the host.

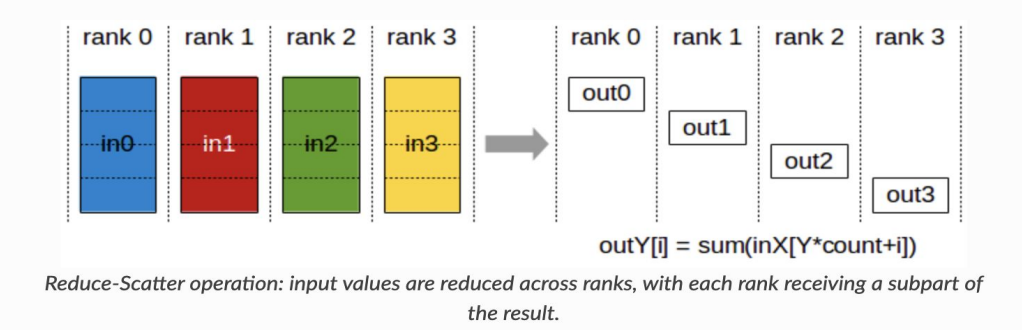
Source: https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/examples.html

Example NCCL AllReduce - 1 GPU per CPU Process

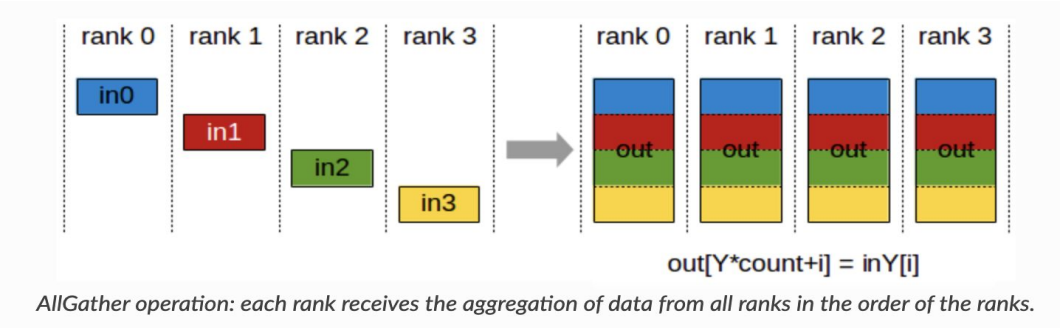
```
//get NCCL unique ID at rank 0 and broadcast it to all others
if (myRank == 0) ncclGetUniqueId(&id);
MPICHECK(MPI_Bcast((void *)&id, sizeof(id), MPI_BYTE, 0, MPI_COMM_WORLD));


//picking a GPU based on localRank, allocate device buffers
CUDACHECK(cudaSetDevice(localRank));
CUDACHECK(cudaMalloc(&sendbuff, size * sizeof(float)));
CUDACHECK(cudaMalloc(&recvbuff, size * sizeof(float)));
CUDACHECK(cudaStreamCreate(&s));


//initializing NCCL
NCCLCHECK(ncclCommInitRank(&comm, nRanks, id, myRank));


//communicating using NCCL
NCCLCHECK(ncclAllReduce((const void*)sendbuff, (void*)recvbuff, size, ncclFloat, ncclSum,
      comm, s));


//completing NCCL operation by synchronizing on the CUDA stream
CUDACHECK(cudaStreamSynchronize(s));
```

Source: https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/examples.html

Ring AllReduce (ReduceScatter + AllGather)



Reduce-Scatter operation: input values are reduced across ranks, with each rank receiving a subpart of the result.



AllGather operation: each rank receives the aggregation of data from all ranks in the order of the ranks.

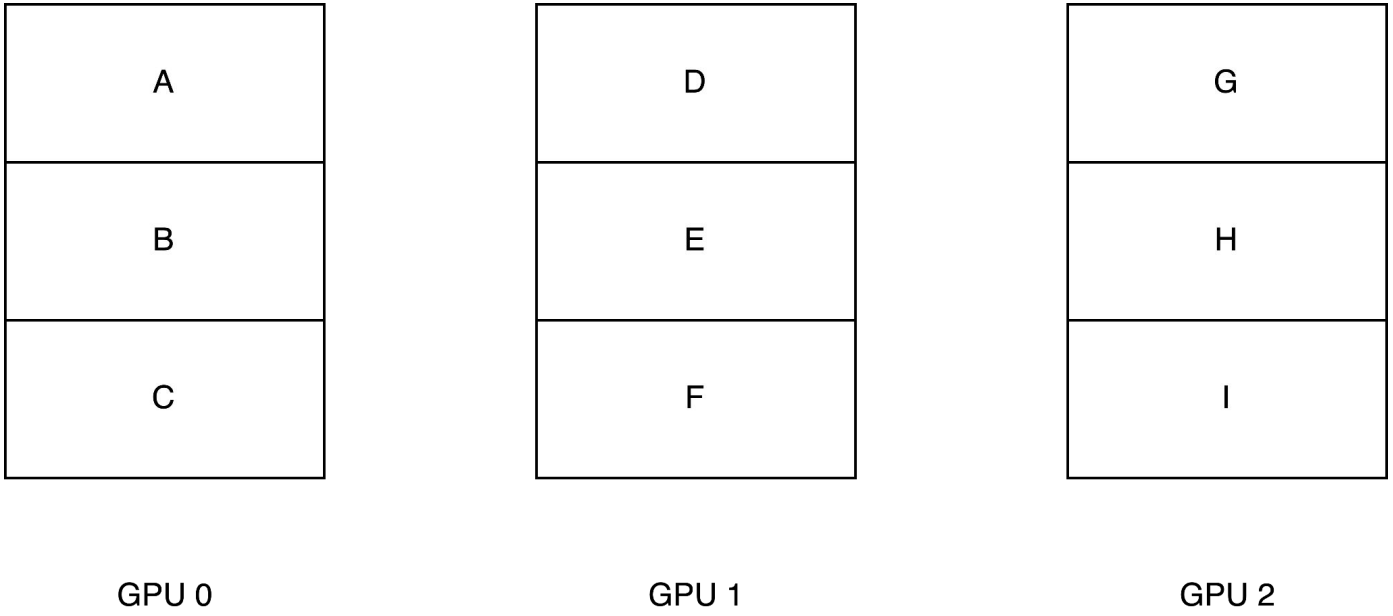
Ring AllReduce (ReduceScatter + AllGather)

```
template<typename T, typename RedOp, typename Proto>
__device__ __forceinline__ void runRing(ncclWorkElem *args) {
  const int tid = threadIdx.x;
  const int nthreads = args->nWarps*WARP_SIZE;
  const int bid = args->bid;
  const int nChannels = args->nChannels;
  ncclRing *ring = &ncclShmem.channel.ring;
  int ringIx = ring->index;
  const ssize_t chunkSize = int(Proto::calcBytePerStep()/sizeof(T) * (Proto::Id == NCCL_PROTO_SIMPLE ? ALLREDUCE_CHUNKSTEPS : 1));
  const int nranks = ncclShmem.comm.nRanks;
  const ssize_t loopSize = nChannels*nranks*chunkSize;
  const ssize_t size = args->count;

  int minChunkSize;
  if (Proto::Id == NCCL_PROTO_LL)
    minChunkSize = nthreads*(Proto::calcBytePerGrain()/sizeof(T));
  if (Proto::Id == NCCL_PROTO_LL128) {
    // We should not need the final /2 but it makes performance much, much smoother. Might be a bug somewhere.
    minChunkSize = nthreads*(Proto::calcBytePerGrain()/sizeof(T))/2;
  }

  Primitives<T, RedOp, FanSymmetric<1>, 1, Proto, 0> prims
    (tid, nthreads, &ring->prev, &ring->next, args->sendbuff, args->recvbuff, args->redOpArg);
```

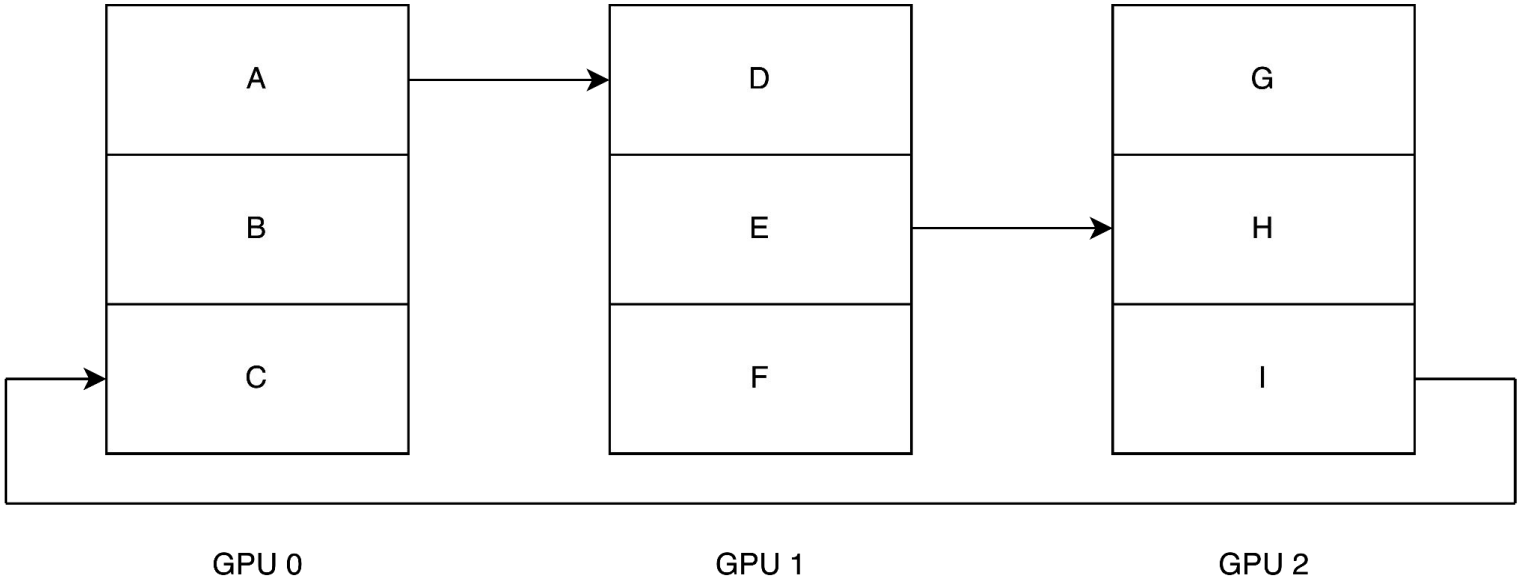
Ring AllReduce (ReduceScatter + AllGather)

```
for (ssize_t gridOffset = 0; gridOffset < size; gridOffset += loopSize) {
  ssize_t realChunkSize;
  if (Proto::Id == NCCL_PROTO_SIMPLE) {
    realChunkSize = min(chunkSize, divUp(size-gridOffset, nChannels*nranks));
    realChunkSize = roundUp(realChunkSize, (nthreads-WARP_SIZE)*sizeof(uint64_t)/sizeof(T));
  }
  else
    realChunkSize = min(chunkSize, divUp(size-gridOffset, nChannels*nranks*minChunkSize)*minChunkSize);
  realChunkSize = int(realChunkSize);

  auto calcOffset = [&]__device__(int chunk)->ssize_t {
    if (Proto::Id == NCCL_PROTO_SIMPLE)
      return gridOffset + bid*nranks*realChunkSize + chunk*realChunkSize;
    else
      return gridOffset + (chunk*nChannels + bid)*realChunkSize;
  };
  auto modRanks = [&]__device__(int r)->int {
    return r - (r >= nranks ? nranks : 0);
  };

  ssize_t offset;
  int nelem;
  int chunk;

  // step 0: push data to next GPU
  chunk = modRanks(ringIx + nranks-1);
  offset = calcOffset(chunk);
  nelem = min(realChunkSize, size-offset);
  prims.send(offset, nelem);
```

Ring AllReduce (ReduceScatter + AllGather)



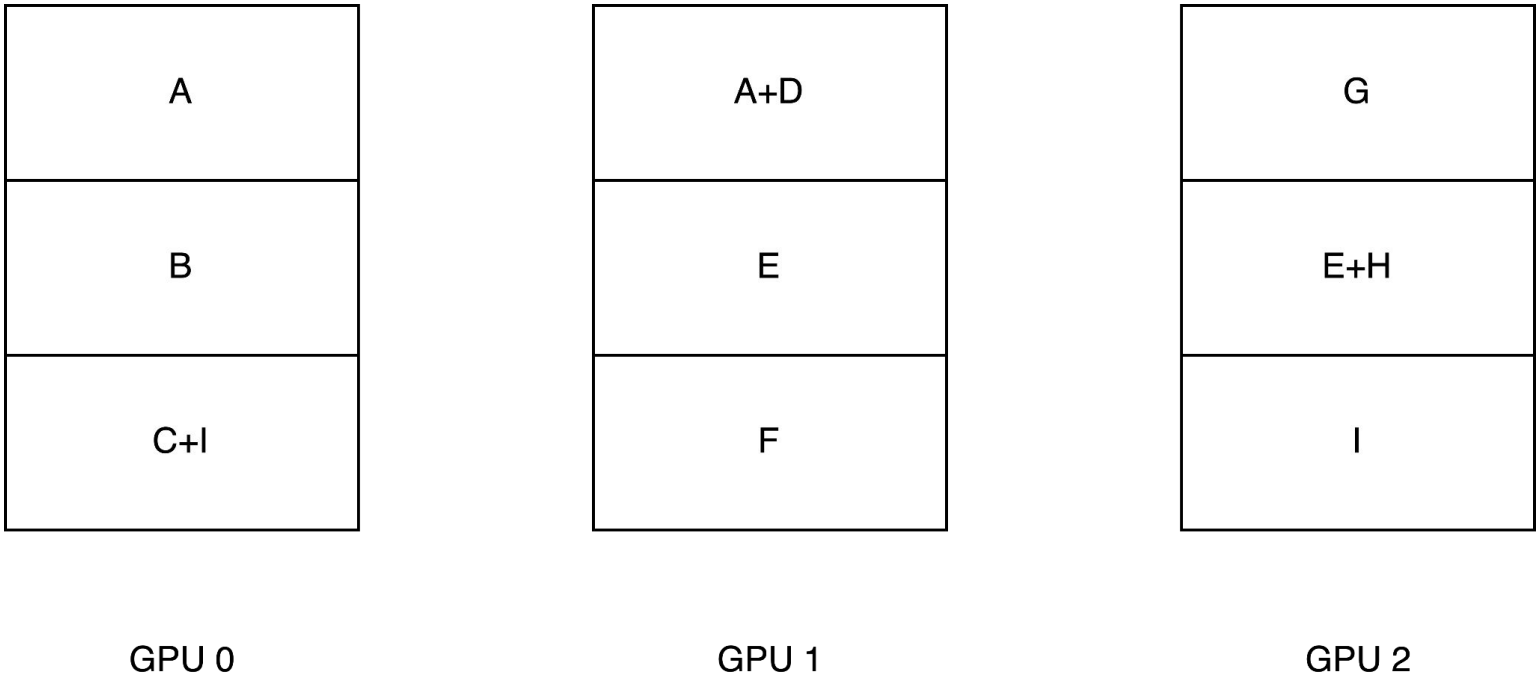
Ring AllReduce (ReduceScatter + AllGather)



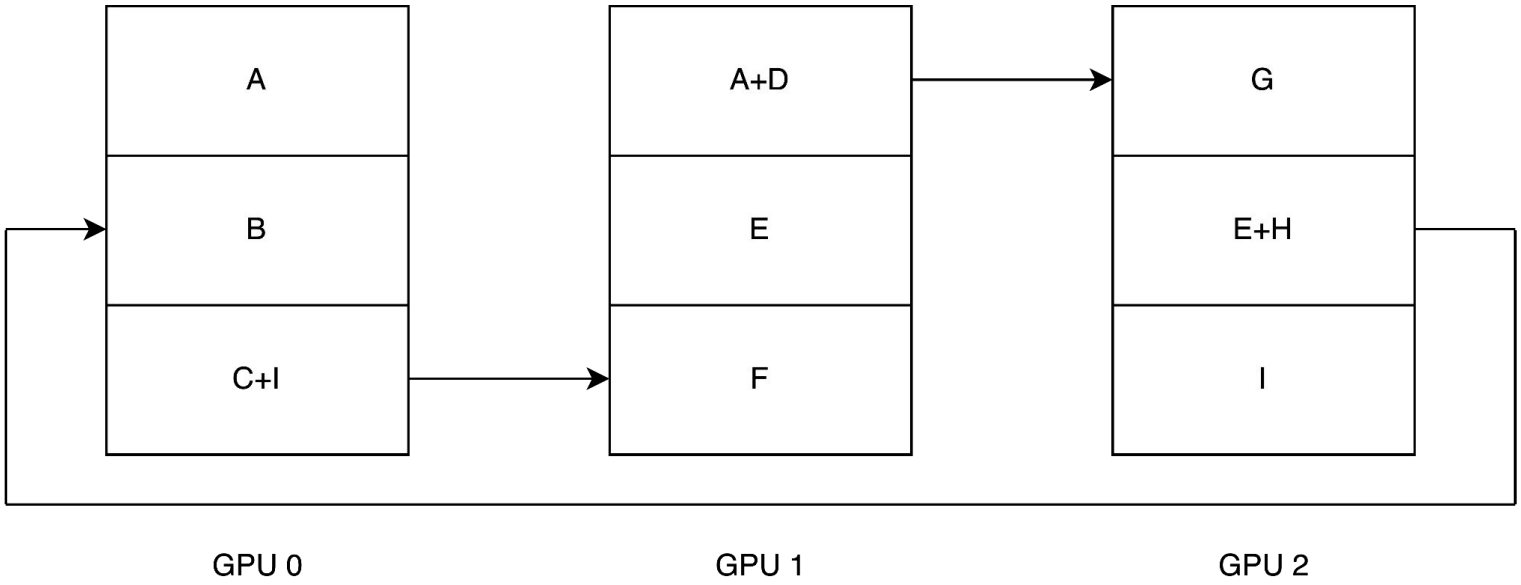
Ring AllReduce (ReduceScatter + AllGather)

```
// k-2 steps: reduce and copy to next GPU
for (int j=2; j<nranks; ++j) {
  chunk = modRanks(ringIx + nranks-j);
  offset = calcOffset(chunk);
  nelem = min(realChunkSize, size-offset);
  prims.recvReduceSend(offset, nelem);
}
```

Ring AllReduce (ReduceScatter + AllGather)



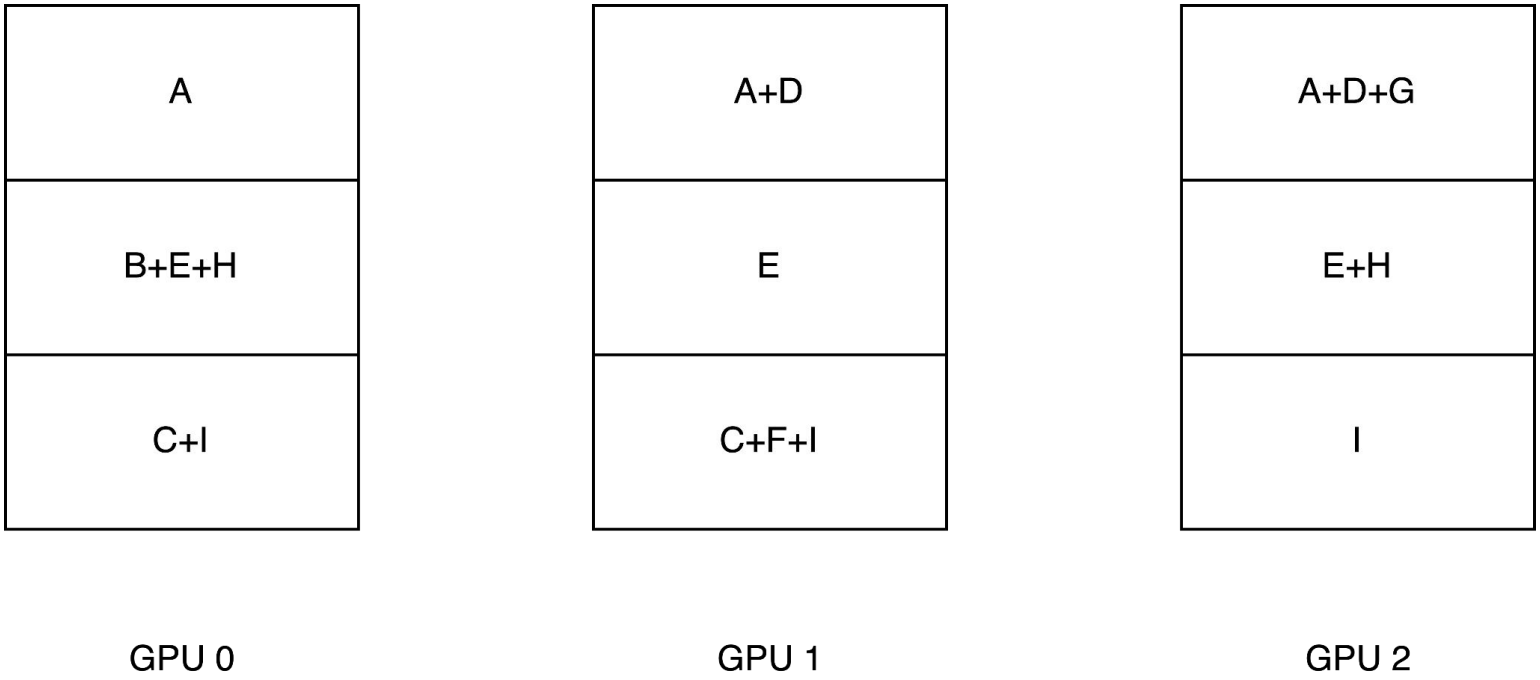
Ring AllReduce (ReduceScatter + AllGather)



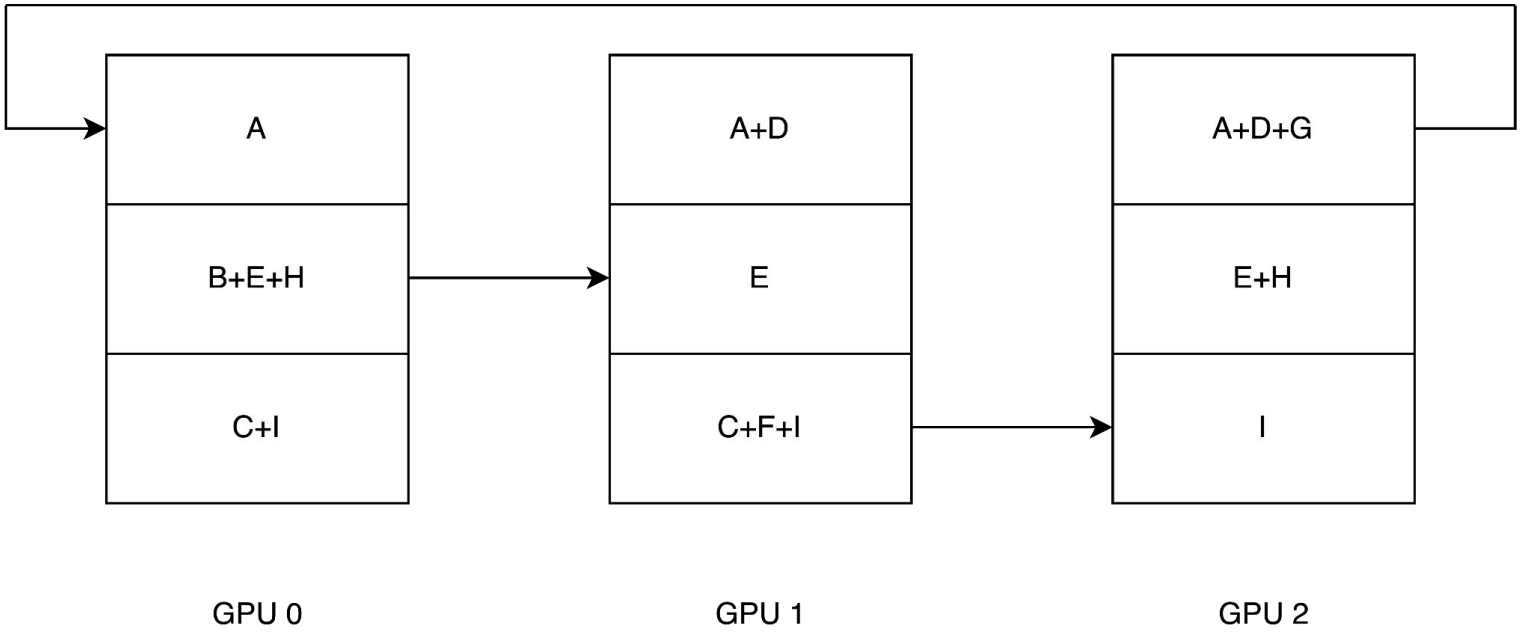
Ring AllReduce (ReduceScatter + AllGather)

```
// step k-1: reduce this buffer and data, which will produce the final
// result that we store in this data and push to the next GPU
chunk = ringIx + 0;
offset = calcOffset(chunk);
nelem = min(realChunkSize, size-offset);
prims.directRecvReduceCopySend(offset, offset, nelem, /*postOp=*/true);
```

Ring AllReduce (ReduceScatter + AllGather)



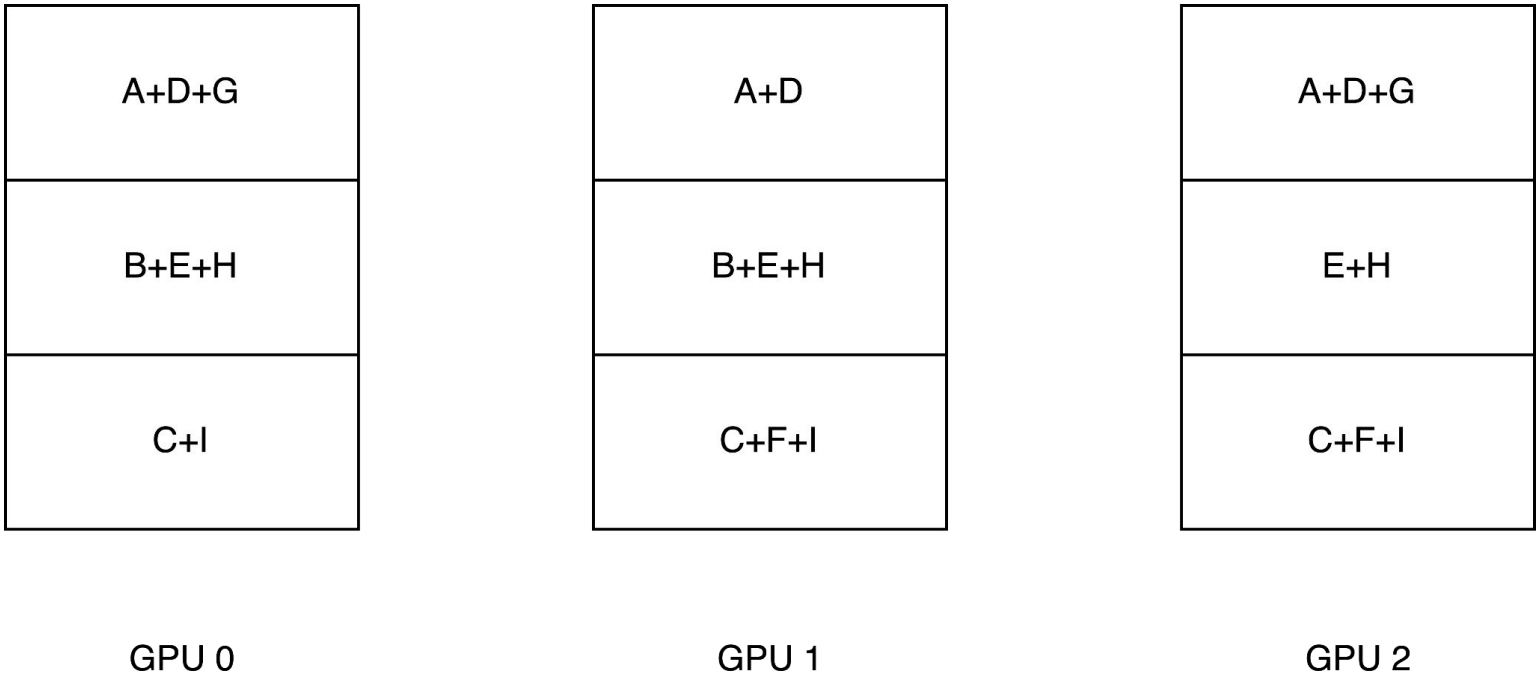
Ring AllReduce (ReduceScatter + AllGather)



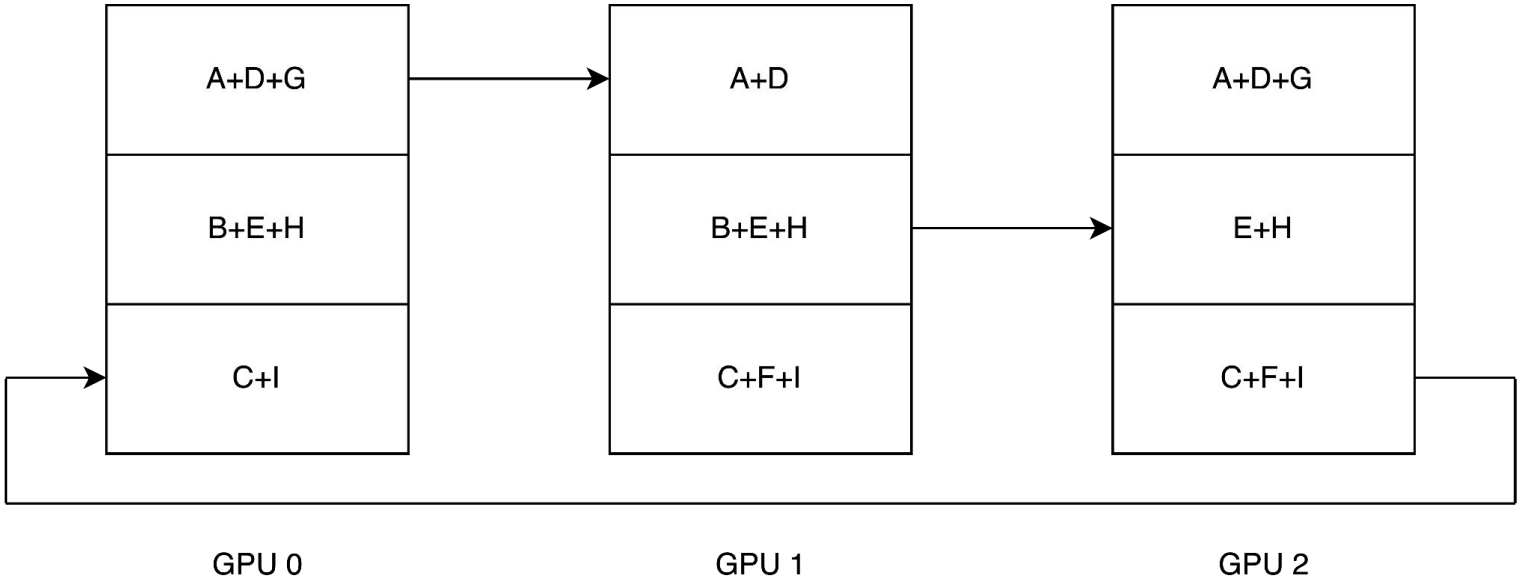
Ring AllReduce (ReduceScatter + AllGather)

```
// k-2 steps: copy to next GPU
for (int j=1; j<nranks-1; ++j) {
  chunk = modRanks(ringIx + nranks-j);
  offset = calcOffset(chunk);
  nelem = min(realChunkSize, size-offset);
  prims.directRecvCopySend(offset, nelem);
}

// Make final copy from buffer to dest.
chunk = modRanks(ringIx + 1);
offset = calcOffset(chunk);
nelem = min(realChunkSize, size-offset);
prims.directRecv(offset, nelem);
```

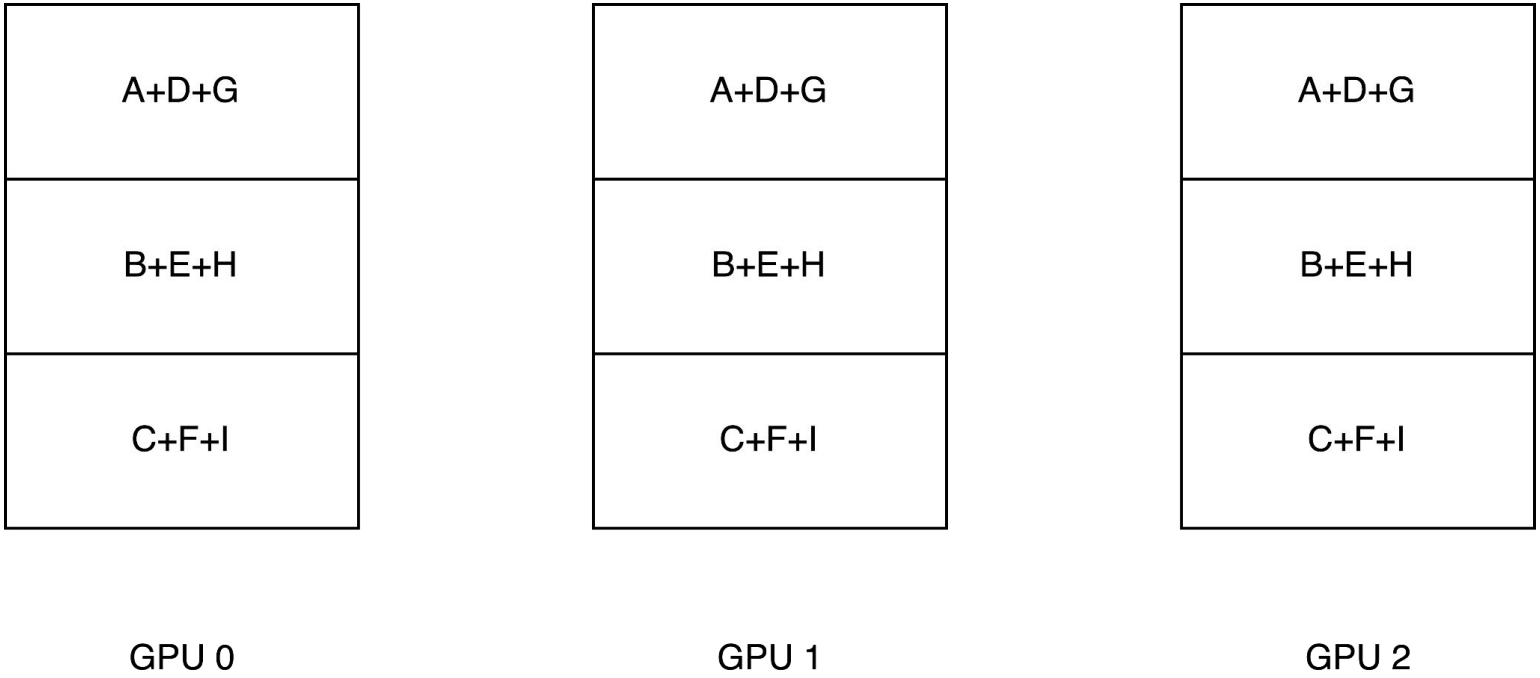
Ring AllReduce (ReduceScatter + AllGather)



Ring AllReduce (ReduceScatter + AllGather)



Ring AllReduce (ReduceScatter + AllGather)



Note: ring traversed in one direction.

Other interesting topics

- Other AllReduce Algorithms
  - Tree AllReduce
- Other Collectives
- Network Topology
  - NVLink
  - Infiniband/RoCE
    - https://network.nvidia.com/pdf/whitepapers/Intro_to_IB_for_End_Users.pdf
  - IP
- Collective Operation Primitives

Tree AllReduce (Reduce + Broadcast)

```
{ // Reduce : max number of recv is 3, max number of send is 1 (binary tree + local)
  Primitives<T, RedOp, FanAsymmetric<NCCL_MAX_TREE_ARITY, 1>, /*Direct=*/0, Proto, 0> prims
    (tid, nthreads, tree->down, &tree->up, args->sendbuff, args->recvbuff, args->redOpArg);
  if (tree->up == -1) {
    for (ssize_t gridOffset = 0; gridOffset < size; gridOffset += loopSize) {
      ssize_t offset = gridOffset + bid*int(chunkSize);
      int nelem = min(chunkSize, size-offset);
      prims.recvReduceCopy(offset, offset, nelem, /*postOp=*/true);
    }
  }
  else if (tree->down[0] == -1) {
    for (ssize_t gridOffset = 0; gridOffset < size; gridOffset += loopSize) {
      ssize_t offset = gridOffset + bid*int(chunkSize);
      int nelem = min(chunkSize, size-offset);
      prims.send(offset, nelem);
    }
  }
  else {
    for (ssize_t gridOffset = 0; gridOffset < size; gridOffset += loopSize) {
      ssize_t offset = gridOffset + bid*int(chunkSize);
      int nelem = min(chunkSize, size-offset);
      prims.recvReduceSend(offset, nelem);
    }
  }
}
```

```
{ // Broadcast : max number of recv is 1, max number of send is 3 (binary tree + local)
  Primitives<T, RedOp, FanAsymmetric<1, NCCL_MAX_TREE_ARITY>, /*Direct=*/1, Proto, 0> prims
    (tid, nthreads, &tree->up, tree->down, args->sendbuff, args->recvbuff, args->redOpArg);
  if (tree->up == -1) {
    for (ssize_t gridOffset = 0; gridOffset < size; gridOffset += loopSize) {
      ssize_t offset = gridOffset + bid*int(chunkSize);
      int nelem = min(chunkSize, size-offset);
      prims.directSendFromOutput(offset, nelem);
    }
  }
  else if (tree->down[0] == -1) {
    for (ssize_t gridOffset = 0; gridOffset < size; gridOffset += loopSize) {
      ssize_t offset = gridOffset + bid*int(chunkSize);
      int nelem = min(chunkSize, size-offset);
      prims.directRecv(offset, nelem);
    }
  }
  else {
    for (ssize_t gridOffset = 0; gridOffset < size; gridOffset += loopSize) {
      ssize_t offset = gridOffset + bid*int(chunkSize);
      int nelem = min(chunkSize, size-offset);
      prims.directRecvCopySend(offset, nelem);
    }
  }
}
```

[NVIDIA Blog about different algorithms]

Collective Operations Prims

- Prims functions like prims.send, prims.recvReduceSend, etc. are how data is sent between GPUs during collective operations
- Implemented in three different “protocols” with different synchronization
  - Simple
  - LL (low latency, 8 byte stores are atomic, 4 bytes of data and 4 bytes of flag)
  - LL128 (low latency, 128 byte stores are atomic, 120 bytes of data and 8 bytes of flag)
- AllReduce has 3 algorithms and 3 protocols for a total of 9 ways it can run