[NCCL](index.html)

[2.25](https://docs.nvidia.com/deeplearning/sdk/nccl-archived/index.html)

    * [ncclGetLastError](api/comms.html#ncclgetlasterror)
    * [ncclGetErrorString](api/comms.html#ncclgeterrorstring)
    * [ncclGetVersion](api/comms.html#ncclgetversion)
    * [ncclGetUniqueId](api/comms.html#ncclgetuniqueid)
    * [ncclCommInitRank](api/comms.html#ncclcomminitrank)
    * [ncclCommInitAll](api/comms.html#ncclcomminitall)
    * [ncclCommInitRankConfig](api/comms.html#ncclcomminitrankconfig)
    * [ncclCommInitRankScalable](api/comms.html#ncclcomminitrankscalable)
    * [ncclCommSplit](api/comms.html#ncclcommsplit)
    * [ncclCommFinalize](api/comms.html#ncclcommfinalize)
    * [ncclCommDestroy](api/comms.html#ncclcommdestroy)
    * [ncclCommAbort](api/comms.html#ncclcommabort)
    * [ncclCommGetAsyncError](api/comms.html#ncclcommgetasyncerror)
    * [ncclCommCount](api/comms.html#ncclcommcount)
    * [ncclCommCuDevice](api/comms.html#ncclcommcudevice)
    * [ncclCommUserRank](api/comms.html#ncclcommuserrank)
    * [ncclCommRegister](api/comms.html#ncclcommregister)
    * [ncclCommDeregister](api/comms.html#ncclcommderegister)
    * [ncclMemAlloc](api/comms.html#ncclmemalloc)
    * [ncclMemFree](api/comms.html#ncclmemfree)
  * [Collective Communication Functions](api/colls.html)
    * [ncclAllReduce](api/colls.html#ncclallreduce)
    * [ncclBroadcast](api/colls.html#ncclbroadcast)
    * [ncclReduce](api/colls.html#ncclreduce)
    * [ncclAllGather](api/colls.html#ncclallgather)
    * [ncclReduceScatter](api/colls.html#ncclreducescatter)
  * [Group Calls](api/group.html)

  * [Values accepted](env.html#id3)

* [NCCL_SOCKET_RETRY_SLEEP_MSEC](env.html#nccl-socket-retry-sleep-msec)

  * [Values accepted](env.html#id4)

* [NCCL_SOCKET_NTHREADS](env.html#nccl-socket-nthreads)

  * [Values accepted](env.html#id5)

* [NCCL_NSOCKS_PERTHREAD](env.html#nccl-nsocks-perthread)

  * [Values accepted](env.html#id6)

* [NCCL_CROSS_NIC](env.html#nccl-cross-nic)

  * [Values accepted](env.html#id7)

* [NCCL_IB_HCA](env.html#nccl-ib-hca)

  * [Values accepted](env.html#id8)

* [NCCL_IB_TIMEOUT](env.html#nccl-ib-timeout)

  * [Values accepted](env.html#id9)

* [NCCL_IB_RETRY_CNT](env.html#nccl-ib-retry-cnt)

  * [Values accepted](env.html#id10)

* [NCCL_IB_GID_INDEX](env.html#nccl-ib-gid-index)

  * [Values accepted](env.html#id11)

* [NCCL_IB_ADDR_FAMILY](env.html#nccl-ib-addr-family)

  * [Values accepted](env.html#id12)

* [NCCL_IB_ADDR_RANGE](env.html#nccl-ib-addr-range)

  * [Values accepted](env.html#id13)

* [NCCL_IB_ROCE_VERSION_NUM](env.html#nccl-ib-roce-version-num)

  * [Values accepted](env.html#id14)

* [NCCL_IB_SL](env.html#nccl-ib-sl)

  * [Values accepted](env.html#id15)

* [NCCL_IB_TC](env.html#nccl-ib-tc)

  * [Values accepted](env.html#id16)

* [NCCL_IB_FIFO_TC](env.html#nccl-ib-fifo-tc)
  * [Values accepted](env.html#id17)
* [NCCL_IB_RETURN_ASYNC_EVENTS](env.html#nccl-ib-return-async-events)
  * [Values accepted](env.html#id18)
* [NCCL_OOB_NET_ENABLE](env.html#nccl-oob-net-enable)
  * [Values accepted](env.html#id19)
* [NCCL_OOB_NET_IFNAME](env.html#nccl-oob-net-ifname)
  * [Values accepted](env.html#id20)
* [NCCL_UID_STAGGER_THRESHOLD](env.html#nccl-uid-stagger-threshold)
  * [Values accepted](env.html#id21)
* [NCCL_UID_STAGGER_RATE](env.html#nccl-uid-stagger-rate)
  * [Values accepted](env.html#id22)
* [NCCL_NET](env.html#nccl-net)
  * [Values accepted](env.html#id23)
* [NCCL_NET_PLUGIN](env.html#nccl-net-plugin)
  * [Values accepted](env.html#id24)
* [NCCL_TUNER_PLUGIN](env.html#nccl-tuner-plugin)
  * [Values accepted](env.html#id25)
* [NCCL_PROFILER_PLUGIN](env.html#nccl-profiler-plugin)
  * [Values accepted](env.html#id26)
* [NCCL_IGNORE_CPU_AFFINITY](env.html#nccl-ignore-cpu-affinity)
  * [Values accepted](env.html#id27)
* [NCCL_CONF_FILE](env.html#nccl-conf-file)
  * [Values accepted](env.html#id28)
* [NCCL_DEBUG](env.html#nccl-debug)
  * [Values accepted](env.html#id30)
* [NCCL_DEBUG_FILE](env.html#nccl-debug-file)

* [NCCL_MAX_NCHANNELS](env.html#nccl-max-nchannels)
  * [Values accepted](env.html#id43)
* [NCCL_MIN_NCHANNELS](env.html#nccl-min-nchannels)
  * [Values accepted](env.html#id44)
* [NCCL_CHECKS_DISABLE](env.html#nccl-checks-disable)
  * [Values accepted](env.html#id45)
* [NCCL_CHECK_POINTERS](env.html#nccl-check-pointers)
  * [Values accepted](env.html#id46)
* [NCCL_LAUNCH_MODE](env.html#nccl-launch-mode)
  * [Values accepted](env.html#id47)
* [NCCL_IB_DISABLE](env.html#nccl-ib-disable)
  * [Values accepted](env.html#id48)
* [NCCL_IB_AR_THRESHOLD](env.html#nccl-ib-ar-threshold)
  * [Values accepted](env.html#id49)
* [NCCL_IB_QPS_PER_CONNECTION](env.html#nccl-ib-qps-per-connection)
  * [Values accepted](env.html#id50)
* [NCCL_IB_SPLIT_DATA_ON_QPS](env.html#nccl-ib-split-data-on-qps)
  * [Values accepted](env.html#id51)
* [NCCL_IB_CUDA_SUPPORT](env.html#nccl-ib-cuda-support)
  * [Values accepted](env.html#id52)
* [NCCL_IB_PCI_RELAXED_ORDERING](env.html#nccl-ib-pci-relaxed-ordering)
  * [Values accepted](env.html#id53)
* [NCCL_IB_ADAPTIVE_ROUTING](env.html#nccl-ib-adaptive-routing)
  * [Values accepted](env.html#id54)
* [NCCL_IB_ECE_ENABLE](env.html#nccl-ib-ece-enable)
  * [Values accepted](env.html#id55)
* [NCCL_MEM_SYNC_DOMAIN](env.html#nccl-mem-sync-domain)

  * [Values accepted](env.html#id56)
* [NCCL_CUMEM_ENABLE](env.html#nccl-cumem-enable)
  * [Values accepted](env.html#id57)
* [NCCL_CUMEM_HOST_ENABLE](env.html#nccl-cumem-host-enable)
  * [Values accepted](env.html#id58)
* [NCCL_NET_GDR_LEVEL (formerly NCCL_IB_GDR_LEVEL)](env.html#nccl-net-gdr-level-formerly-nccl-ib-gdr-level)
  * [Values accepted](env.html#id59)
  * [Integer Values (Legacy)](env.html#id60)
* [NCCL_NET_GDR_READ](env.html#nccl-net-gdr-read)
  * [Values accepted](env.html#id61)
* [NCCL_NET_SHARED_BUFFERS](env.html#nccl-net-shared-buffers)
  * [Value accepted](env.html#id62)
* [NCCL_NET_SHARED_COMMS](env.html#nccl-net-shared-comms)
  * [Value accepted](env.html#id63)
* [NCCL_SINGLE_RING_THRESHOLD](env.html#nccl-single-ring-threshold)
  * [Values accepted](env.html#id64)
* [NCCL_LL_THRESHOLD](env.html#nccl-ll-threshold)
  * [Values accepted](env.html#id65)
* [NCCL_TREE_THRESHOLD](env.html#nccl-tree-threshold)
  * [Values accepted](env.html#id66)
* [NCCL_ALGO](env.html#nccl-algo)
  * [Values accepted](env.html#id67)
* [NCCL_PROTO](env.html#nccl-proto)
  * [Values accepted](env.html#id68)
* [NCCL_NVB_DISABLE](env.html#nccl-nvb-disable)
  * [Value accepted](env.html#id69)

* [NCCL_PXN_DISABLE](env.html#nccl-pxn-disable)
  * [Value accepted](env.html#id70)
* [NCCL_P2P_PXN_LEVEL](env.html#nccl-p2p-pxn-level)
  * [Value accepted](env.html#id71)
* [NCCL_RUNTIME_CONNECT](env.html#nccl-runtime-connect)
  * [Value accepted](env.html#id72)
* [NCCL_GRAPH_REGISTER](env.html#nccl-graph-register)
  * [Value accepted](env.html#id74)
* [NCCL_LOCAL_REGISTER](env.html#nccl-local-register)
  * [Value accepted](env.html#id75)
* [NCCL_LEGACY_CUDA_REGISTER](env.html#nccl-legacy-cuda-register)
  * [Value accepted](env.html#id76)
* [NCCL_SET_STACK_SIZE](env.html#nccl-set-stack-size)
  * [Value accepted](env.html#id77)
* [NCCL_GRAPH_MIXING_SUPPORT](env.html#nccl-graph-mixing-support)
  * [Value accepted](env.html#id79)
* [NCCL_DMABUF_ENABLE](env.html#nccl-dmabuf-enable)
  * [Value accepted](env.html#id80)
* [NCCL_P2P_NET_CHUNKSIZE](env.html#nccl-p2p-net-chunksize)
  * [Values accepted](env.html#id81)
* [NCCL_P2P_LL_THRESHOLD](env.html#nccl-p2p-ll-threshold)
  * [Values accepted](env.html#id82)
* [NCCL_ALLOC_P2P_NET_LL_BUFFERS](env.html#nccl-alloc-p2p-net-ll-buffers)
  * [Values accepted](env.html#id83)
* [NCCL_COMM_BLOCKING](env.html#nccl-comm-blocking)
  * [Values accepted](env.html#id84)
* [NCCL_CGA_CLUSTER_SIZE](env.html#nccl-cga-cluster-size)

* * *

ExamplesÂ¶

The examples in this section provide an overall view of how to use NCCL in various environments, combining one or multiple techniques:

* using multiple GPUs per thread/process
* using multiple threads

* using multiple processes - the examples with multiple processes use MPI as parallel runtime environment, but any multi-process system should be able to work similarly.

Ensure that you always check the return codes from the NCCL functions. For clarity, the following examples do not contain error checking.

Communicator Creation and Destruction ExamplesÂ¶

The following examples demonstrate common use cases for NCCL initialization.

Example 1: Single Process, Single Thread, Multiple DevicesÂ¶

In the specific case of a single process, ncclCommInitAll can be used. Here is an example creating a communicator for 4 devices, therefore, there are 4 communicator objects:

```
ncclComm_t comms[4];
int devs[4] = { 0, 1, 2, 3 };
ncclCommInitAll(comms, 4, devs);
```

Next, you can call NCCL collective operations using a single thread and group calls, or multiple threads, each provided with a comm object.

At the end of the program, all of the communicator objects are destroyed:

```
  for (int i=0; i<4; i++)
    ncclCommDestroy(comms[i]);
```

The following code depicts a complete working example with a single process that manages multiple devices:

```
  #include <stdlib.h>
  #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)



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((void**)sendbuff + i, size * sizeof(float)));
      CUDACHECK(cudaMalloc((void**)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));
    }


    //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]));
    }



    //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;
  }
```

Example 2: One Device per Process or ThreadÂ¶

When a process or host thread is responsible for at most one GPU,

ncclCommInitRank can be used as a collective call to create a communicator.

Each thread or process will get its own object.

The following code is an example of a communicator creation in the context of MPI, using one device per MPI rank.

First, we retrieve MPI information about processes:

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

Next, a single rank will create a unique ID and send it to all other ranks to make sure everyone has it:

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

Finally, we create the communicator:

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

We can now call the NCCL collective operations using the communicator.

```
ncclAllReduce( ... , comm);
```

Finally, we destroy the communicator object:

```
ncclCommDestroy(comm);
```

The following code depicts a complete working example with multiple MPI processes and one device per process:

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

```
#include <unistd.h>
#include <stdint.h>
#include <stdlib.h>


#define MPICHECK(cmd) do {                          \
  int e = cmd;                                      \
  if( e != MPI_SUCCESS ) {                          \
    printf("Failed: MPI error %s:%d '%d'\n",        \
        __FILE__,__LINE__, e);   \
    exit(EXIT_FAILURE);                             \
  }                                                 \
} while(0)


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


#define NCCLCHECK(cmd) do {                         \
  ncclResult_t r = cmd;                             \
```

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



static uint64_t getHostHash(const char* string) {
  // Based on DJB2a, result = result * 33 ^ char
  uint64_t result = 5381;
  for (int c = 0; string[c] != '\0'; c++){
    result = ((result << 5) + result) ^ string[c];
  }
  return result;
}



static void getHostName(char* hostname, int maxlen) {
  gethostname(hostname, maxlen);
  for (int i=0; i< maxlen; i++) {
    if (hostname[i] == '.') {
        hostname[i] = '\0';
        return;
    }
  }
}
```

```
int main(int argc, char* argv[])
{
  int size = 32*1024*1024;


  int myRank, nRanks, localRank = 0;


  //initializing MPI
  MPICHECK(MPI_Init(&argc, &argv));
  MPICHECK(MPI_Comm_rank(MPI_COMM_WORLD, &myRank));
  MPICHECK(MPI_Comm_size(MPI_COMM_WORLD, &nRanks));


  //calculating localRank based on hostname which is used in selecting a GPU
  uint64_t hostHashs[nRanks];
  char hostname[1024];
  getHostName(hostname, 1024);
  hostHashs[myRank] = getHostHash(hostname);
        MPICHECK(MPI_Allgather(MPI_IN_PLACE,  0,  MPI_DATATYPE_NULL,  hostHashs,
sizeof(uint64_t), MPI_BYTE, MPI_COMM_WORLD));
  for (int p=0; p<nRanks; p++) {
    if (p == myRank) break;
    if (hostHashs[p] == hostHashs[myRank]) localRank++;
  }
```

```
ncclUniqueId id;
ncclComm_t comm;
float *sendbuff, *recvbuff;
cudaStream_t s;


//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));



    //free device buffers
    CUDACHECK(cudaFree(sendbuff));
    CUDACHECK(cudaFree(recvbuff));



    //finalizing NCCL
    ncclCommDestroy(comm);



    //finalizing MPI
    MPICHECK(MPI_Finalize());



    printf("[MPI Rank %d] Success \n", myRank);
    return 0;
  }
```

Example 3: Multiple Devices per ThreadÂ¶

You can combine both multiple process or threads and multiple device per

process or thread. In this case, we need to use group semantics.

The following example combines MPI and multiple devices per process (=MPI rank).

First, we retrieve MPI information about processes:

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

Next, a single rank will create a unique ID and send it to all other ranks to make sure everyone has it:

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

Then, we create our ngpus communicator objects, which are part of a larger group of ngpus*nRanks:

```
ncclComm_t comms[ngpus];
ncclGroupStart();
for (int i=0; i<ngpus; i++) {
  cudaSetDevice(devs[i]);
  ncclCommInitRank(comms+i, ngpus*nRanks, id, myRank*ngpus+i);
}
ncclGroupEnd();
```

Next, we call NCCL collective operations using a single thread and group calls, or multiple threads, each provided with a comm object.

At the end of the program, we destroy all communicators objects:

```
for (int i=0; i<ngpus; i++)
  ncclCommDestroy(comms[i]);
```

The following code depicts a complete working example with multiple MPI processes and multiple devices per process:

```
#include <stdio.h>
#include "cuda_runtime.h"
#include "nccl.h"
#include "mpi.h"
#include <unistd.h>
#include <stdint.h>


#define MPICHECK(cmd) do {                          \
  int e = cmd;                                      \
  if( e != MPI_SUCCESS ) {                          \
    printf("Failed: MPI error %s:%d '%d'\n",        \
        __FILE__,__LINE__, e);   \
    exit(EXIT_FAILURE);                             \
  }                                                 \
} while(0)


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

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


static uint64_t getHostHash(const char* string) {
  // Based on DJB2a, result = result * 33 ^ char
  uint64_t result = 5381;
  for (int c = 0; string[c] != '\0'; c++){
    result = ((result << 5) + result) ^ string[c];
  }
  return result;
}


static void getHostName(char* hostname, int maxlen) {
  gethostname(hostname, maxlen);
  for (int i=0; i< maxlen; i++) {
    if (hostname[i] == '.') {
        hostname[i] = '\0';
        return;
```

```
    }
   }
  }

  int main(int argc, char* argv[])
  {
   int size = 32*1024*1024;


   int myRank, nRanks, localRank = 0;


   //initializing MPI
   MPICHECK(MPI_Init(&argc, &argv));
   MPICHECK(MPI_Comm_rank(MPI_COMM_WORLD, &myRank));
   MPICHECK(MPI_Comm_size(MPI_COMM_WORLD, &nRanks));


   //calculating localRank which is used in selecting a GPU
   uint64_t hostHashs[nRanks];
   char hostname[1024];
   getHostName(hostname, 1024);
   hostHashs[myRank] = getHostHash(hostname);
            MPICHECK(MPI_Allgather(MPI_IN_PLACE,  0,  MPI_DATATYPE_NULL,  hostHashs,
sizeof(uint64_t), MPI_BYTE, MPI_COMM_WORLD));
   for (int p=0; p<nRanks; p++) {
```

```
    if (p == myRank) break;
    if (hostHashs[p] == hostHashs[myRank]) localRank++;
}



//each process is using two GPUs
int nDev = 2;



float** sendbuff = (float**)malloc(nDev * sizeof(float*));
float** recvbuff = (float**)malloc(nDev * sizeof(float*));
cudaStream_t* s = (cudaStream_t*)malloc(sizeof(cudaStream_t)*nDev);



//picking GPUs based on localRank
for (int i = 0; i < nDev; ++i) {
  CUDACHECK(cudaSetDevice(localRank*nDev + 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));
}



ncclUniqueId id;
ncclComm_t comms[nDev];
```

```
    //generating NCCL unique ID at one process and broadcasting it to all
    if (myRank == 0) ncclGetUniqueId(&id);
    MPICHECK(MPI_Bcast((void *)&id, sizeof(id), MPI_BYTE, 0, MPI_COMM_WORLD));


    //initializing NCCL, group API is required around ncclCommInitRank as it is
    //called across multiple GPUs in each thread/process
    NCCLCHECK(ncclGroupStart());
    for (int i=0; i<nDev; i++) {
      CUDACHECK(cudaSetDevice(localRank*nDev + i));
      NCCLCHECK(ncclCommInitRank(comms+i, nRanks*nDev, id, myRank*nDev + i));
    }
    NCCLCHECK(ncclGroupEnd());


    //calling NCCL communication API. Group API is required when using
    //multiple devices per thread/process
    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 stream to complete NCCL communication
  for (int i=0; i<nDev; i++)
      CUDACHECK(cudaStreamSynchronize(s[i]));



  //freeing device memory
  for (int i=0; i<nDev; i++) {
     CUDACHECK(cudaFree(sendbuff[i]));
     CUDACHECK(cudaFree(recvbuff[i]));
  }



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



  //finalizing MPI
  MPICHECK(MPI_Finalize());



  printf("[MPI Rank %d] Success \n", myRank);
  return 0;
}
```

Example 4: Multiple communicators per deviceÂ¶

NCCL allows users to create multiple communicators per device. The following code shows an example with multiple MPI processes, one device per process, and multiple communicators per device:

```
  // blocking communicators
  CUDACHECK(cudaSetDevice(localRank));
  for (int i = 0; i < commNum; ++i) {
    if (myRank == 0) ncclGetUniqueId(&id);
    MPICHECK(MPI_Bcast((void *)&id, sizeof(id), MPI_BYTE, 0, MPI_COMM_WORLD));
    NCCLCHECK(ncclCommInitRank(&blockingComms[i], nRanks, id, myRank));
  }

  // non-blocking communicators
  CUDACHECK(cudaSetDevice(localRank));
  ncclConfig_t config = NCCL_CONFIG_INITIALIZER;
  config.blocking = 0;
  for (int i = 0; i < commNum; ++i) {
    if (myRank == 0) ncclGetUniqueId(&id);
    MPICHECK(MPI_Bcast((void *)&id, sizeof(id), MPI_BYTE, 0, MPI_COMM_WORLD));
    NCCLCHECK(ncclCommInitRankConfig(&nonblockingComms[i], nRanks, id, myRank, &config));
    do {
      NCCLCHECK(ncclCommGetAsyncError(nonblockingComms[i], &state));
    } while(state == ncclInProgress && checkTimeout() != true);
```

```
}
```

checkTimeout() should be a user-defined function. For more nonblocking communicator usage, please check [Fault Tolerance](usage/communicators.html#ft). In addition, if you want to split communicators instead of creating a new one, please check [`ncclCommSplit()`](api/comms.html#c.ncclCommSplit "ncclCommSplit").

Communication ExamplesÂ¶

The following examples demonstrate common patterns for executing NCCL collectives.

Example 1: One Device per Process or ThreadÂ¶

If you have a thread or process per device, then each thread calls the collective operation for its device, for example, AllReduce:

```
ncclAllReduce(sendbuff, recvbuff, count, datatype, op, comm, stream);
```

After the call, the operation has been enqueued to the stream. Therefore, you can call cudaStreamSynchronize if you want to wait for the operation to be complete:

```
cudaStreamSynchronize(stream);
```

For a complete working example with MPI and single device per MPI process, see â€œExample 2: One Device per Process or Threadâ€•.

Example 2: Multiple Devices per ThreadÂ¶

When a single thread manages multiple devices, you need to use group semantics to launch the operation on multiple devices at once:

```
ncclGroupStart();
for (int i=0; i<ngpus; i++)
  ncclAllReduce(sendbuffs[i], recvbuff[i], count, datatype, op, comms[i], streams[i]);
ncclGroupEnd();
```

After ncclGroupEnd, all of the operations have been enqueued to the stream. Therefore, you can now call cudaStreamSynchronize if you want to wait for the operation to be complete:

```
    for (int i=0; i<ngpus; i++)
      cudaStreamSynchronize(streams[i]);
```

For a complete working example with MPI and multiple devices per MPI process, see Example 3: Multiple Devices per Thread.

[Next](mpi.html "NCCL and MPI") [Previous](nccl1.html "Migrating from NCCL 1 to NCCL 2")

* * *