NVIDIA COLLECTIVE COMMUNICATION LIBRARY (NCCL)

DU-08527-204_v01 | July 2017

Developer Guide

TABLE OF CONTENTS

Chapter 1.
OVERVIEW

The NVIDIA® Collective Communications Library (NCCL™, pronounced "Nickel") is a library of multi-GPU collective communication primitives that are topology-aware and can be easily integrated into applications.

Collective communication algorithms employ many processors working in concert to aggregate data. NCCL is not a full-blown parallel programming framework; rather, it is a library focused on accelerating collective communication primitives. The following collective operations are currently supported:

▶ **AllReduce**
▶ **Broadcast**
▶ **Reduce**
▶ **AllGather**
▶ **ReduceScatter**

Tight synchronization between communicating processors is a key aspect of collective communication. CUDA® based collectives would traditionally be realized through a combination of CUDA memory copy operations and CUDA kernels for local reductions. NCCL, on the other hand, implements each collective in a single kernel handling both communication and computation operations. This allows for fast synchronization and minimizes the resources needed to reach peak bandwidth.

NCCL conveniently removes the need for developers to optimize their applications for specific machines. NCCL provides fast collectives over multiple GPUs both within and across nodes. It supports a variety of interconnect technologies including PCIe, NVLink™, InfiniBand Verbs, and IP sockets. NCCL also automatically patterns its communication strategy to match the system's underlying GPU interconnect topology.

Next to performance, ease of programming was the primary consideration in the design of NCCL. NCCL uses a simple C API, which can be easily accessed from a variety of programming languages. NCCL closely follows the popular collectives API defined by MPI (Message Passing Interface). Anyone familiar with MPI will thus find NCCL's API very natural to use. In a minor departure from MPI, NCCL collectives take a "stream" argument which provides direct integration with the CUDA programming model. Finally, NCCL is compatible with virtually any multi-GPU parallelization model, for example:

- ▸ single-threaded
- ▸ multi-threaded, for example, using one thread per GPU
- ▸ multi-process, for example, MPI combined with multi-threaded operation on GPUs

NCCL has found great application in Deep Learning Frameworks, where the `AllReduce` collective is heavily used for neural network training. Efficient scaling of neural network training is possible with the multi-GPU and multi node communication provided by NCCL.

Chapter 2.
COLLECTIVE COMMUNICATION PRIMITIVES

Collective communication primitives are common patterns of data transfer among a group of CUDA devices. A communication algorithm involves many processors that are communicating together.

Each NCCL processor (GPU) is identified within the communication group by zero-based index or rank . Each *rank* uses a communicator object to refer to the collection of GPUs that are intended to work together for some task.

The creation of a communicator is the first step needed before launching any communication operation.

2.1. Creating a Communicator

When creating a communicator, a unique rank between `0` and `n-1` has to be assigned to each of the `n` CUDA devices which are part of the communicator.

Given a static mapping of ranks to CUDA devices, the ncclCommInitRank and ncclCommInitAll functions will create communicator objects, each communicator object being associated to a fixed rank. Those objects will then be used to launch communication operations.

> Before calling `ncclCommInitRank`, you need to first create a unique object which will be used by all processes and threads to synchronize and understand they are part of the same communicator. This is done by calling the `ncclGetUniqueId` function.
>
> The `ncclGetUniqueId` function returns an ID which has to be broadcast to all participating threads and processes using any CPU communication system, for example, passing the ID pointer to multiple threads, or broadcasting it to other processes using MPI or another parallel environment using, for example, sockets.

You can also call the `ncclCommInitAll` function to create `n` communicator objects at once within a single process. As it is limited to a single process, this function does not permit inter-node communication. `ncclCommInitAll` is equivalent to calling a combination of `ncclGetUniqueId` and `ncclCommInitRank`.

The following sample code is a simplified implementation of `ncclCommInitAll`:

```
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();
}
```

2.2. Operations

Like MPI collective operations, NCCL collective operations have to be called for each rank (hence CUDA device) to form a complete collective operation. Failure to do so will result in other ranks waiting indefinitely.

2.2.1. `AllReduce`

The `AllReduce` operation is performing reductions on data, for example, sum and max, across devices and writing the result in the receive buffers of every rank.

The `AllReduce` operation is rank-agnostic. Any reordering of the ranks will not affect the outcome of the operations.

AllReduce starts with independent arrays V_k of N values on each of K ranks and ends with identical arrays S of N values, where $S[i] = V_0[i]+V_1[i]+...+V_{k-1}[i]$, for each rank k.
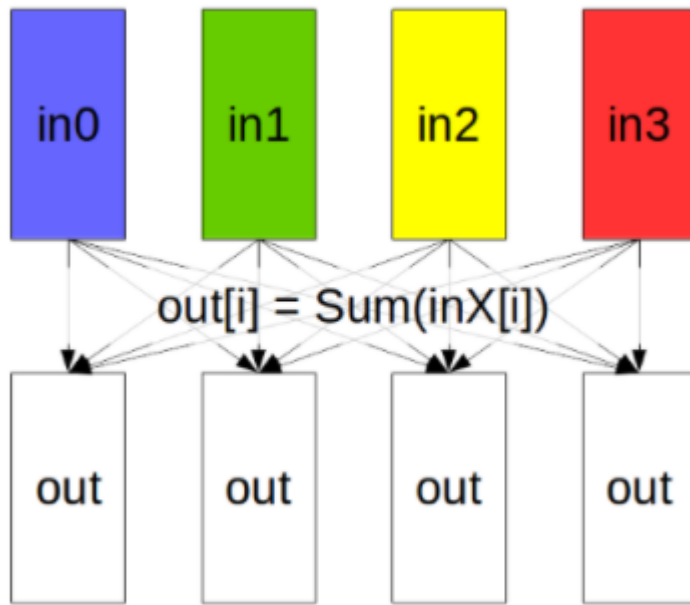
Figure 1   `AllReduce` operation: each rank receives the reduction of input values across ranks.

2.2.2. `Broadcast`

The `Broadcast` operation copies an **N**-element buffer on the root rank to all other ranks.
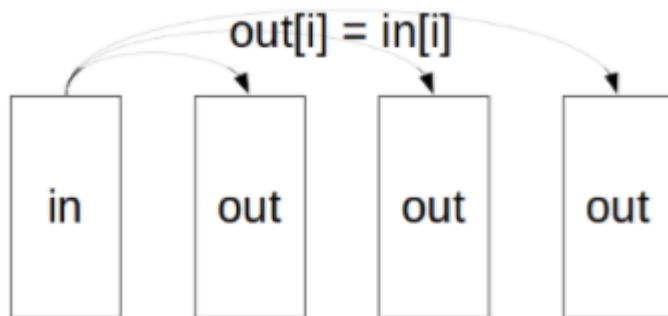

Figure 2   `Broadcast` operation: all ranks receive data from a root rank.

> **Important** The `root` argument is one of the ranks, not a device number, and is therefore impacted by a different rank to device mapping.

2.2.3. `Reduce`

The `Reduce` operation is performing the same operation as `AllReduce`, but writes the result only in the receive buffers of a specified root rank.
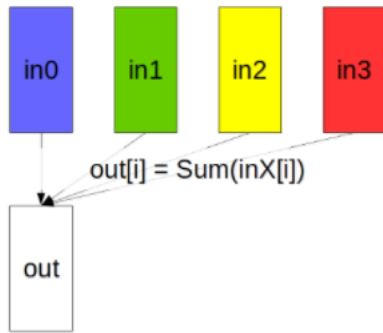
Figure 3 `Reduce` operation: one rank receives the reduction of input values across ranks.

> **Important** The `root` argument is one of the ranks, not a device number, and is therefore impacted by a different rank to device mapping.

> A **Reduce**, followed by a **Broadcast**, is equivalent to the **AllReduce** operation.

2.2.4. AllGather

In the **AllGather** operation, each of the **K** processors aggregates **N** values from every processor into an output of dimension **K*N**. The output is ordered by rank index.
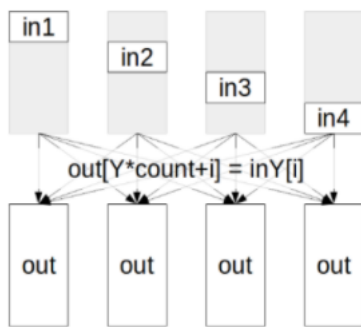

Figure 4 `AllGather` operation: each rank receives the aggregation of data from all ranks in the order of the ranks.

The **AllGather** operation is impacted by a different rank or device mapping since the ranks determine the data layout.

> Executing **ReduceScatter**, followed by **AllGather**, is equivalent to the **AllReduce** operation.

2.2.5. ReduceScatter

The **ReduceScatter** operation performs the same operation as the **Reduce** operation, except the result is scattered in equal blocks among ranks, each rank getting a chunk of data based on its rank index.
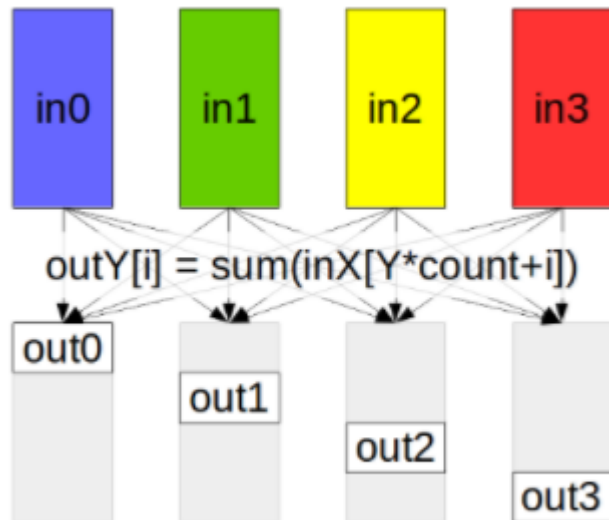
Figure 5 `ReduceScatter` operation: input values are reduced across ranks, with each rank receiving a sub-part of the result.

The **ReduceScatter** operation is impacted by a different rank or device mapping since the ranks determine the data layout.

2.3. Data Pointers

In general NCCL will accept any CUDA pointers that are accessible from the CUDA device associated to the communicator object. This includes:

▸ device memory local to the CUDA device
▸ host memory registered using CUDA SDK APIs `cudaHostRegister` or `cudaGetDevicePointer`
▸ managed and unified memory

The only exception is device memory located on another device but accessible from the current device using peer access. NCCL will return an error in that case to avoid programming errors.

2.4. CUDA Stream Semantics

NCCL calls are associated to a stream and are passed as the last argument of the collective communication function. The NCCL call returns when the operation has been

effectively enqueued to the given stream, or returns an error. The collective operation is then executed asynchronously on the CUDA device. The operation status can be queried using standard CUDA semantics, for example, calling `cudaStreamSynchronize` or using CUDA events.

2.5. Group Calls

When a single thread is managing multiple devices, group semantics must be used. This is because every NCCL call may have to block, waiting for other threads or ranks to arrive, before effectively posting the NCCL operation on the given stream.

Hence, a simple loop on multiple devices like shown below could block on the first call waiting for the other ones:

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

To define that these calls are part of the same collective operation, use the `ncclGroupStart` and `ncclGroupEnd` functions. For example:

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

This will tell NCCL to treat all calls between `ncclGroupStart` and `ncclGroupEnd` as a single call to many devices.

> **Caution** When called inside a group, `ncclAllReduce` can return without having enqueued the operation on the stream. Stream operations like `cudaStreamSynchronize` can therefore be called only after `ncclGroupEnd` returns.

Contrary to NCCL 1.x, there is no need to set the CUDA device before every NCCL communication call within a group, but it is still needed when calling `ncclCommInitRank` within a group.

2.6. Thread Safety

NCCL primitives are generally not thread-safe, however, they are reentrant. Multiple threads should use separate communicator objects.

2.7. In-place Operations

Contrary to MPI, NCCL does not define a special "in-place" value to replace pointers. Instead, NCCL optimizes the case where the provided pointers are effectively "in place".

For **ncclReduce** and **ncclAllreduce** functions, this means that passing **sendBuff == recvBuff** will perform in place operations, storing final results at the same place as initial data was read from.

For **ncclReduceScatter** and **ncclAllGather**, in place operations are done when the per-rank pointer is located at the rank offset of the global buffer. More precisely, these calls are considered in place:

```
ncclReduceScatter(data, data+rank*recvcount, recvcount, datatype,
 op, comm, stream);
ncclAllGather(data+rank*sendcount, data, sendcount, datatype, op,
 comm, stream);
```

Chapter 3.
DOWNLOADING NCCL 2.0.4

To download NCCL 2.0.4, see NVIDIA Collective Communications Library (NCCL) home page.

3.1. Software Requirements

Ensure your environment meets the following software requirements:

▶ NCCL release 2.0 supports Ubuntu 14.04 and 16.04.
▶ CUDA 8.0 for NvLink and PCIe systems

3.2. Hardware Requirements

NCCL supports all CUDA devices with a compute capability of 3.0 and higher. For the compute capability of all NVIDIA GPUs, check: CUDA GPUs.

Chapter 4.
USING NCCL 2.0.4

Using NCCL is similar to using any other library in your code. For example:

1. Install the NCCL library onto your system.
   For more information, see Downloading NCCL.
2. Modify your application to link to that library.
3. Include the header file `nccl.h` in your application.
4. Create a communicator.
   For more information, see Creating a Communicator.
5. Familiarize yourself with the NCCL API documentation to maximize your usage performance.
   For more information, see the *NCCL API* documentation.

Chapter 5.
MIGRATING FROM NCCL 1.X TO NCCL 2.0

If you are using NCCL 1.x and want to move to NCCL 2.0, be aware that the APIs have changed slightly. NCCL 2.0 supports all of the collectives that NCCL 1.x supports, but with slight modifications to the API.

In addition, NCCL 2.0 also requires the usage of the Group API when a single thread manages NCCL calls for multiple GPUs.

The following list summarizes the changes that may be required in usage of NCCL API when using an application has a single thread that manages NCCL calls for multiple GPUs, and is ported from NCCL 1.x to 2.0:

Initialization

In 1.x, NCCL had to be initialized using `ncclCommInitAll` at a single thread or having one thread per GPU concurrently call `ncclCommInitRank`. NCCL 2.0 retains these two modes of initialization. It adds a new mode with the Group API where `ncclCommInitRank` can be called in a loop, like a communication call, as shown below. The loop has to be guarded by the Group start and stop API.

```
ncclGroupStart();
 for (int i=0; i<ngpus; i++) {
   cudaSetDevice(i);
   ncclCommInitRank(comms+i, ngpus, id, i);
 }
 ncclGroupEnd();
```

Communication

In NCCL 2.0, the collective operation can be initiated for different devices by making calls in a loop, on a single thread. This is similar to the usage in NCCL 1.x. However, this loop has to be guarded by the Group API in 2.0. Unlike in 1.x, the application does not have to select the relevant CUDA device before making the communication API call. NCCL runtime internally selects the device associated with the NCCL communicator handle. For example:

```
ncclGroupStart();
 for (int i=0; i<nLocalDevs; i++) {
```

```
    ncclAllReduce(..., comm[i], stream[i];
 }
 ncclGroupEnd();
```

When using only one device per thread or one device per process, the general usage of API remains unchanged from NCCL 1.x to 2.0. Group API is not required in this case.

Counts

Counts provided as arguments are now of type **`size_t`** instead of **`integer`**.

In-place usage for AllGather and ReduceScatter

For more information, see In-place Operations.

AllGather arguments order

The AllGather function had its arguments reordered. The prototype changed from:

```
ncclResult_t  ncclAllGather(const void* sendbuff, int count,
 ncclDataType_t datatype,
    void* recvbuff, ncclComm_t comm, cudaStream_t stream);
```

to:

```
ncclResult_t  ncclAllGather(const void* sendbuff, void*
 recvbuff, size_t sendcount,
    ncclDataType_t datatype, ncclComm_t comm, cudaStream_t
 stream);
```

The recvbuff argument has been moved after the sendbuff argument to be consistent with all the other operations.

Datatypes

New datatypes have been added in NCCL 2.0. The ones present in NCCL 1.x did not change and are still usable in NCCL 2.x.

Error codes

Error codes have been merged into the **`ncclInvalidArgument`** category and have been simplified. A new **`ncclInvalidUsage`** code has been created to cover new programming errors.

Chapter 6.
EXAMPLES

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.

6.1. Communicator Creation and Destruction Examples

The following examples demonstrate common use cases for NCCL initialization.

6.1.1. Example 1: Single Process, Single Thread, Multiple Devices

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

#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)

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));
  }

  //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;
}
```

6.1.2. Example 2: One Device per Process or Thread

When one thread or process is affected to each thread, **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, 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.

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>

#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 DJB2, 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;
}
```

6.1.3. Example 3: Multiple Devices per Thread

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 DJB2, 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
ncclGroupStart();
for (int i=0; i<nDev; i++) {
    CUDACHECK(cudaSetDevice(localRank*nDev + i));
    NCCLCHECK(ncclCommInitRank(comms+i, nRanks*nDev, id,
myRank*nDev + i));
}
ncclGroupEnd();

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

//synchrozing 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;
}
```

6.2. Communication Examples

The following examples demonstrate common patterns for executing NCCL collectives.

6.2.1. Example 1: One Device per Process or Thread

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 Example 2: One Device per Process or Thread.

6.2.2. Example 2: Multiple Devices per Thread

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.

Chapter 7.
NCCL AND MPI

7.1. API

The NCCL API and usage is similar to MPI but there are many minor differences. The following list summarizes these differences:

Using multiple devices per process

Similarly to the concept of MPI endpoints, NCCL does not require ranks to be mapped 1:1 to MPI ranks. A NCCL communicator may have many ranks associated to a single process (hence MPI rank if used with MPI).

ReduceScatter operation

The `ncclReduceScatter` operation is similar to the `MPI_Reduce_scatter_block` operation, not the `MPI_Reduce_scatter` operation. The `MPI_Reduce_scatter` function is intrinsically a "vector" function, while `MPI_Reduce_scatter_block` (defined later to fill the missing semantics) provides regular counts similarly to the mirror function `MPI_Allgather`. This is an oddity of MPI which has not been fixed for legitimate retro-compatibility reasons and that NCCL does not follow.

Send and Receive counts

In many collective operations, MPI allows for different send and receive counts and types, as long as `sendcount*sizeof(sendtype) == recvcount*sizeof(recvtype)`. NCCL does not allow that, defining a single count and a single data-type.

For `AllGather` and `ReduceScatter` operations, the count is equal to the per-rank size, which is the smallest size; the other count being equal to `nranks*count`. The function prototype clearly shows which count is provided, for example:

▶ `sendcount` for `ncclAllgather`

  ▸   **`recvcount`** for **`ncclReduceScatter`**

> 💬 When performing or comparing **`AllReduce`** operations using a combination of **`ReduceScatter`** and **`AllGather`** , define the **`sendcount`** and **`recvcount`** as the total count divided by the number of ranks, with the correct count rounding-up, if it is not a perfect multiple of the number of ranks.

In-place operations

For more information, see In-place Operations.

7.2. Using NCCL within an MPI Program

NCCL can be easily used in conjunction with MPI. NCCL collectives are similar to MPI collectives, therefore, creating a NCCL communicator out of an MPI communicator is straightforward. It is therefore easy to use MPI for CPU-to-CPU communication and NCCL for GPU-to-GPU communication.

However, some implementation details in MPI can lead to issues when using NCCL inside an MPI program.

7.2.1. MPI Progress

MPI defines a notion of *progress* which means that MPI operations need the program to call MPI functions (potentially multiple times) to make progress and eventually complete.

In some implementations, progress on one rank may need MPI to be called on another rank. While this is usually bad for performance, it can be argued that this is a valid MPI implementation.

As a result, blocking in a NCCL collective operations, for example calling `cudaStreamSynchronize`, may create a deadlock in some cases because not calling MPI will not make other ranks progress, hence reach the NCCL call, hence unblock the NCCL operation.

In that case, the `cudaStreamSynchronize` call should be replaced by a loop like the following:

```
cudaError_t err = cudaErrorNotReady;
int flag;
while (err == cudaErrorNotReady) {
  err = cudaStreamQuery(args->streams[i]);
  MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &flag,
MPI_STATUS_IGNORE);
}
```

7.2.2. Inter-GPU Communication with CUDA-aware MPI

Using NCCL to perform inter-GPU communication concurrently with CUDA-aware MPI may create deadlocks.

NCCL creates inter-device dependencies, meaning that after it has been launched, a NCCL kernel will wait (and potentially block the CUDA device) until all ranks in the communicator launch their NCCL kernel. CUDA-aware MPI may also create such dependencies between devices depending on the MPI implementation.

Using both MPI and NCCL to perform transfers between the same sets of CUDA devices concurrently is therefore not guaranteed to be safe.

Chapter 8.
TROUBLESHOOTING

Ensure you are familiar with the following known issues and useful debugging strategies.

8.1. Known Issues

Ensure you are familiar with the following known issues:

Sharing Data

In order to share data between ranks, NCCL may require shared system memory for IPC and pinned (page-locked) system memory resources. The operating system's limits on these resources may need to be increased accordingly. Please see your system's documentation for details. In particular, Docker containers default to limited shared and pinned memory resources. When using NCCL inside a container, it is recommended that you increase these resources by issuing:

```
--shm-size=1g --ulimit memlock=-1
```

in the command line to

```
nvidia-docker run
```

Concurrency between NCCL and CUDA calls

NCCL uses CUDA kernels to perform inter-GPU communication. The NCCL kernels synchronize with each other, therefore, each kernel requires other kernels on other GPUs to be also executed in order to complete. The application should therefore make sure that nothing prevents the NCCL kernels from being executed concurrently on the different devices of a NCCL communicator.

For example, let's say you have a process managing multiple CUDA devices, and, also features a thread which calls CUDA functions asynchronously. In this case, CUDA calls could be executed between the enqueuing of two NCCL kernels. The CUDA call

may wait for the first NCCL kernel to complete and prevent the second one from being launched, causing a deadlock since the first kernel will not complete until the second one is executed. To avoid this issue, one solution is to have a lock around the NCCL launch on multiple devices (around `ncclGroupStart` and `ncclGroupEnd` when using a single thread, around the NCCL launch when using multiple threads, using thread synchronization if necessary) and take this lock when calling CUDA from the asynchronous thread.

8.2. NCCL Knobs

A knob is a type of environment variable that can you can turn on or off by setting specific values. These environment variables should be set in the context of running NCCL. The following table lists all of the available knobs that can be modified in NCCL.
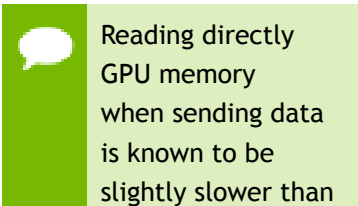
Table 1   Knobs available for modification in NCCL

Environment Variable	Description	Values Accepted
`NCCL_SHM_DISABLE`	The `NCCL_SHM_DISABLE` variable disables the Shared Memory (SHM) transports. SHM is used between devices when peer-to-peer cannot happen, therefore, host memory is used. NCCL uses network (InfiniBand or IP sockets) to communicate between the CPU sockets when SHM is disabled.	Define and set to `1` to disable SHM.
`NCCL_SOCKET_IFNAME`	The `NCCL_SOCKET_IFNAME` variable specifies which IP interface to use for communication. This variable also defines a prefix for the network interfaces to be filtered.	Define and set to `ib` or `eth`. The value searches for all applicable `ib*` or `eth*` named interfaces on the system. Another accepted value is `^eth`, which searches for interfaces that do not match `eth`. Loopback `(lo)` is not selected by NCCL unless it is explicitly set in

Environment Variable	Description	Values Accepted
		the environment variable.

`NCCL_DEBUG`	The `NCCL_DEBUG` variable controls the debug information that is displayed from NCCL. This variable is commonly used for debugging.	**VERSION**
		  Prints the NCCL version at the start of the program.
		WARN
		  Prints an explicit error message whenever any NCCL call errors out.

`NCCL_IB_DISABLE`	The `NCCL_IB_DISABLE` variable disables the IB transport that is to be used by NCCL. Instead, NCCL will fallback to using IP sockets.	Define and set to `1` to force IP sockets usage.
`NCCL_BUFFSIZE`	The NCCL_BUFFSIZE variable controls the amount of buffer to share data between two GPUs. Use this variable if you encounter memory constraint issues when using NCCL or you think that a different buffer size would improve performance.	Default is `4194304` (4 MB). Values are integers, in bytes. The recommendation is to use powers of 2. For example, 1024 will give a 1K buffer.
`NCCL_NTHREADS`	The NCCL_NTHREADS variable sets the number of CUDA threads per CUDA block. NCCL will launch one block per communication ring. Use this variable if you think your GPU clocks are low and you want to increase the number of threads. You can also use this variable to reduce the number of threads to decrease the GPU workload.	▸ Default is `512` for Kepler. ▸ Default is `256` for Maxwell and Pascal. The values allowed are `128`, `256` and `512`.
`NCCL_RINGS`	The NCCL_RINGS variable overrides the rings that NCCL forms by default. Rings are	Ranks from `0` to `n-1`, where `n` is the number of GPUs in your communicator.

Environment Variable	Description	Values Accepted
	sequences of ranks. They can be any permutations of ranks. NCCL filters out any rings that do not contain the number of ranks in the NCCL communicator. In general, the ring formation is dependent on the hardware topology connecting the GPUs in your system.	The ranks can be separated by any non-digit character, for example, " ", "-", except "|". Multiple rings can be specified separated by the pipe character "|". For example, if you have 4 GPUs in a communicator, you can form communication rings as such:`0 1 2 3 | 3 2 1 0`. This will form two rings, one in each direction.
`NCCL_IB_TIMEOUT`	The `NCCL_IB_TIMEOUT` variable controls the InfiniBand Verbs Timeout. Refer to the InfiniBand documentation for more information.	The default value used by NCCL is `14`. The value depends on the size of your InfiniBand network.
`NCCL_IB_CUDA_SUPPORT`	The `NCCL_IB_CUDA_SUPPORT` variable is used to disable GPU Direct RDMA.	By default, NCCL enables GPU Direct RDMA, if the topology permits it. This variable can disable this behavior. Define and set to `0` to disable GPU Direct RDMA.
`NCCL_NET_GDR_READ`	The `NCCL_NET_GDR_READ` variable enables GPU Direct RDMA when sending data. By default, NCCL uses GPU Direct RDMA to receive data directly in GPU memory. However, when sending data, the data is first stored in CPU memory, then goes to the InfiniBand card. Reading directly GPU memory when sending data is known to be slightly slower than	Default value is `0`. Define and set to `1` to use GPU Direct RDMA to send data to the NIC directly (bypassing CPU).

Environment Variable	Description	Values Accepted
	reading from CPU memory.	

8.3. Support

Register for the NVIDIA developer program to report bugs, issues and make requests for feature enhancements. For more information, see: https://developer.nvidia.com/ developer-program.