NVIDIA COLLECTIVE COMMUNICATION LIBRARY (NCCL)

DU-08527-210_v01 | May 2018

Developer Guide

TABLE OF CONTENTS

Chapter 1. Overview.... 1
Chapter 2. Collective Communication Primitives.... 3
2.1. Creating A Communicator.... 3
2.2. Operations.... 4
2.3. Data Pointers.... 7
2.4. CUDA Stream Semantics.... 8
2.5. Group Calls.... 8
2.5.1. Management Of Multiple GPUs From One Thread.... 8
2.5.2. Aggregated Operations (NCCL 2.2 and later).... 9
2.5.2.1. Usage.... 9
2.6. Thread Safety.... 9
2.7. In-Place Operations.... 10
Chapter 3. Examples.... 11
3.1. Communicator Creation And Destruction Examples.... 11
3.1.1. Example 1: Single Process, Single Thread, Multiple Devices.... 11
3.1.2. Example 2: One Device Per Process Or Thread.... 13
3.1.3. Example 3: Multiple Devices Per Thread.... 16
3.2. Communication Examples.... 19
3.2.1. Example 1: One Device Per Process Or Thread.... 19
3.2.2. Example 2: Multiple Devices Per Thread.... 19
Chapter 4. NCCL And MPI.... 21
4.1. API.... 21
4.2. Using NCCL Within An MPI Program.... 22
4.2.1. MPI Progress.... 22
4.2.2. Inter-GPU Communication With CUDA-Aware MPI.... 23
Chapter 5. Troubleshooting.... 24
5.1. Errors.... 24
5.2. Networking Issues.... 24
5.2.1. IP Network Interfaces.... 24
5.2.2. InfiniBand.... 25
5.3. Known Issues.... 25
5.4. NCCL Knobs.... 26
5.5. Support.... 31

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 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. Using the same CUDA device multiple times as different ranks of the same NCCL communicator is not supported and may lead to hangs.

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 $S[i] = V_0[i]+V_1[i]+\ldots+V_{k-1}[i]$, for each rank **k** .

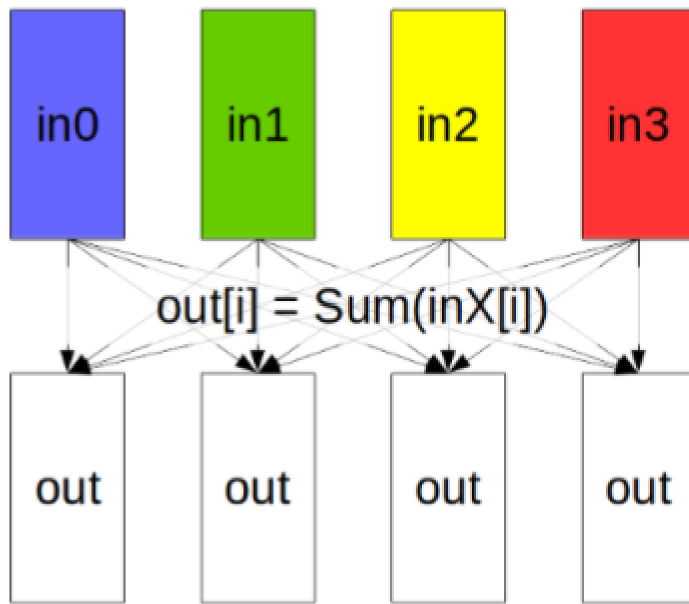


Figure 1 `AllReduce` operation: each rank receives the reduction of input values across ranks.

2.2.2. `Broadcast`

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

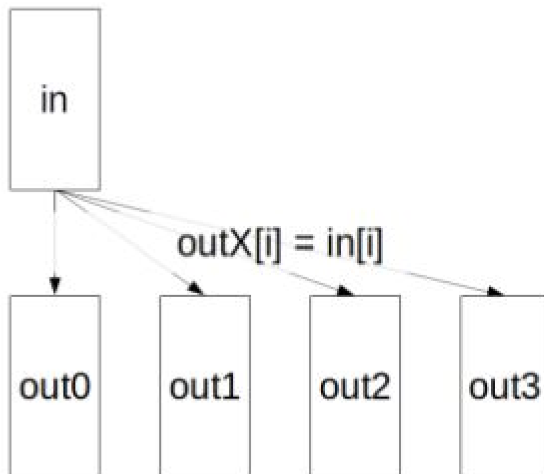


Figure 2 `Broadcast` operation: all ranks receive data from a root rank.

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.

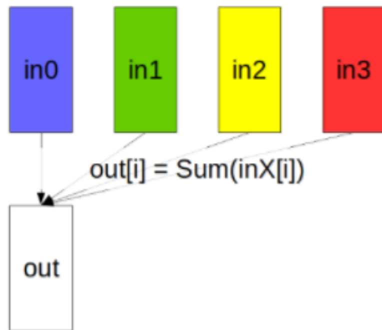


Figure 3 `Reduce` operation: one rank receives the reduction of input values across ranks.

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.

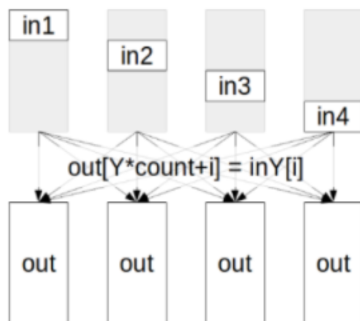
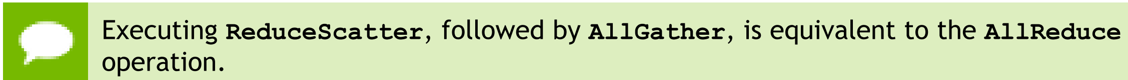


Figure 4 `AllGather` operation: each rank receives the aggregation of data from all ranks in the order of the ranks.

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.

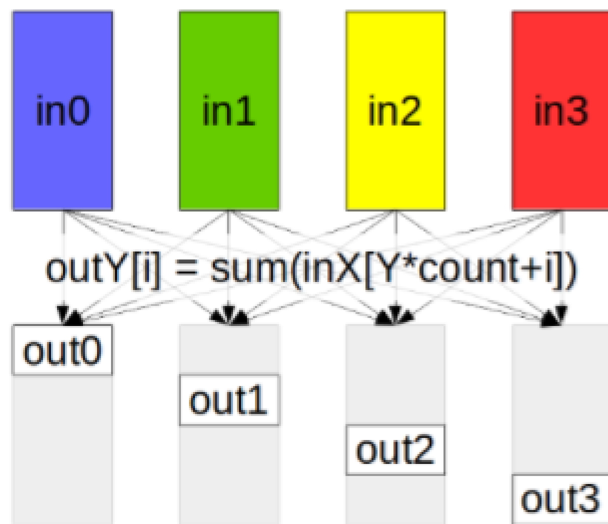


Figure 5 `ReduceScatter` operation: input values are reduced across ranks, with each rank receiving a sub-part of the result.

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 (only when **NCCL_CHECK_POINTERS=1** since 2.2.12).

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

2.5.1. Management Of Multiple GPUs From One Thread

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.

2.5.2. Aggregated Operations (NCCL 2.2 and later)

The group semantics can also be used to have multiple collective operations performed within a single NCCL launch. This is useful for reducing the launch overhead, in other words, latency, as it only occurs once for multiple operations.

2.5.2.1. Usage

Aggregation of collective operations can be done simply by having multiple calls to NCCL within a **ncclGroupStart** / **ncclGroupEnd** section.

In the following example, we launch one broadcast and two **allReduce** operations together as a single NCCL launch.

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

It is not permitted to use different streams for a given NCCL communicator. This sequence is erroneous:

```
ncclGroupStart();
ncclAllReduce(sendbuff1, recvbuff1, count1, comm, stream1);
ncclAllReduce(sendbuff2, recvbuff2, count2, comm, stream2);
ncclGroupEnd();
```

It is, however, permitted to combine aggregation with multi-GPU launch and use different communicators in a group launch as shown in the Management Of Multiple GPUs From One Thread section. When combining multi-GPU launch and aggregation, **ncclGroupStart** and **ncclGroupEnd** can be either used once or at each level. The following example groups the **allReduce** operations from different layers and on multiple CUDA devices:

```
ncclGroupStart();
for (int i=0; i<nlayers; i++) {
  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]);
  }
  ncclGroupEnd();
}
ncclGroupEnd();
```

The NCCL operation will only be started as a whole during the last call to **ncclGroupEnd**. The **ncclGroupStart** and **ncclGroupEnd** calls within the for loop are not necessary and do nothing. Also, a given communicator **comms[g]** is always used with the same stream **streams[g]**.

2.6. Thread Safety

NCCL primitives are generally not thread-safe, however, they are re-entrant. 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 **ncclBroadcast** 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. 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.

3.1. Communicator Creation And Destruction Examples

The following examples demonstrate common use cases for NCCL initialization.

3.1.1. 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;
}
```

3.1.2. 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;
}
```

3.1.3. 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
 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; inDev; 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;
}
```

3.2. Communication Examples

The following examples demonstrate common patterns for executing NCCL collectives.

3.2.1. 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.

3.2.2. 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 4. NCCL AND MPI

4.1. 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.

For more information about the NCCL API, see NCCL API Guide.

4.2. 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.

4.2.1. 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);
}
```

4.2.2. 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 5. TROUBLESHOOTING

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

5.1. Errors

NCCL calls may return a variety of return codes. Ensure that the return codes are always equal to **`ncclSuccess`**. If any call fails, and returns a value different from **`ncclSuccess`**, setting **`NCCL_DEBUG`** to **`WARN`** will make NCCL print an explicit warning message before returning the error.

Errors are grouped into different categories.

- **`ncclUnhandledCudaError`** and **`ncclSystemError`** indicate that a call to an external library failed.
- **`ncclInvalidArgument`** and **`ncclInvalidUsage`** indicates there was a programming error in the application using NCCL.

In either case, refer to the NCCL warning message to understand how to resolve the problem.

5.2. Networking Issues

5.2.1. IP Network Interfaces

NCCL auto-detects which network interfaces to use for inter-node communication. If some interfaces are in state **`up`**, however are not able to communicate between nodes, NCCL may try to use them anyway and therefore fail during the **`init`** functions or even hang.

For more information about how to specify which interfaces to use, see NCCL Knobs topic, particularly the **NCCL_SOCKET_IFNAME** knob.

5.2.2. InfiniBand

Before running NCCL on InfiniBand, running low-level InfiniBand tests (and in particular the **ib_write_bw** test) can help verify which nodes are able to communicate properly.

5.3. 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 up to 2.0.5 or CUDA 8)

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.

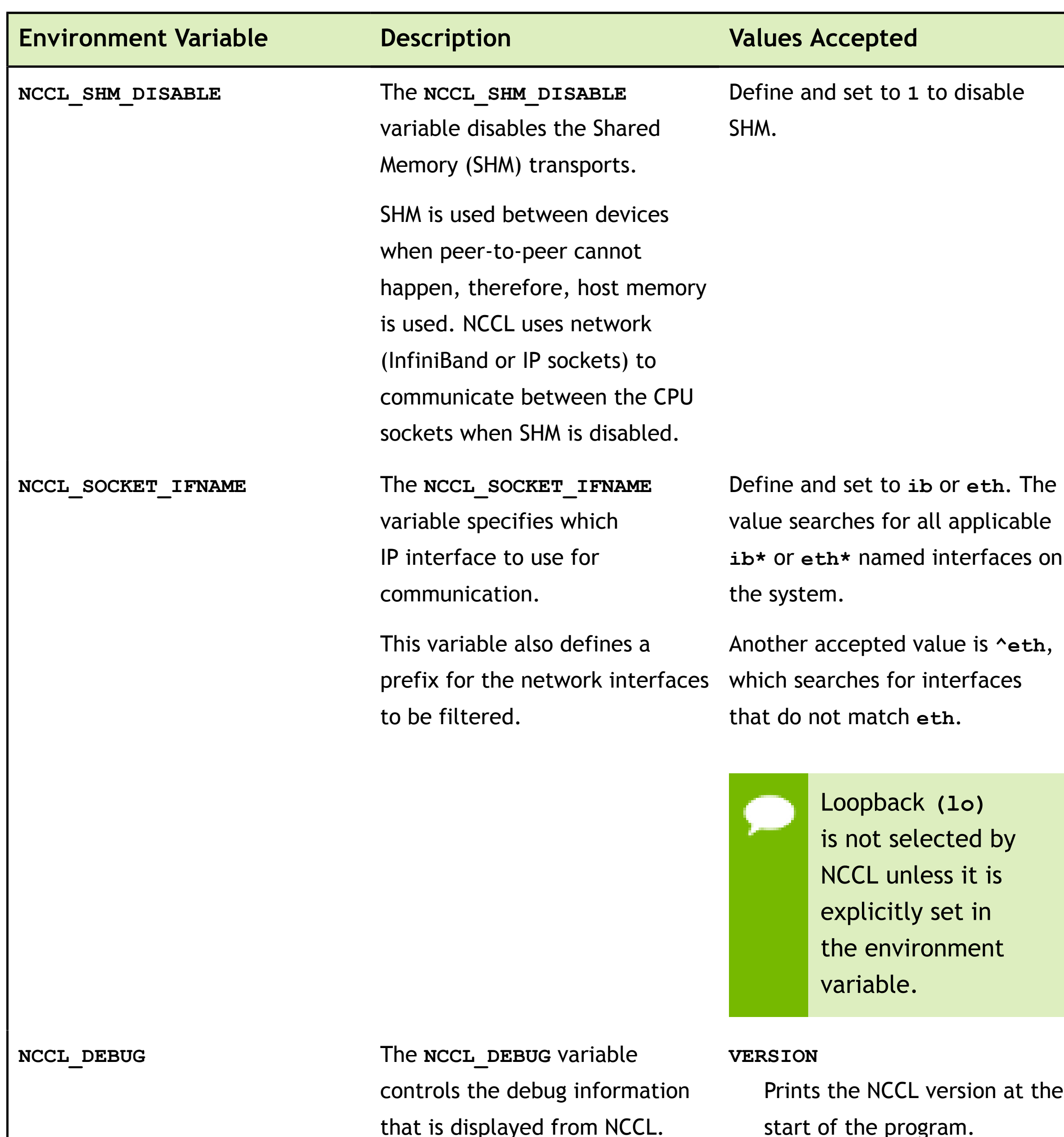
For example, let's say you have a process managing multiple CUDA devices, and, also features a thread which calls CUDA functions asynchronously. In this case, CUDA calls could be executed between the enqueuing of two NCCL kernels. The CUDA call may wait for the first NCCL kernel to complete and prevent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

Starting with NCCL 2.1.0, this issue is no longer present when using CUDA 9, unless **Cooperative Group Launch** is disabled in the **NCCL_LAUNCH_MODE=PARALLEL** setting.

5.4. 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

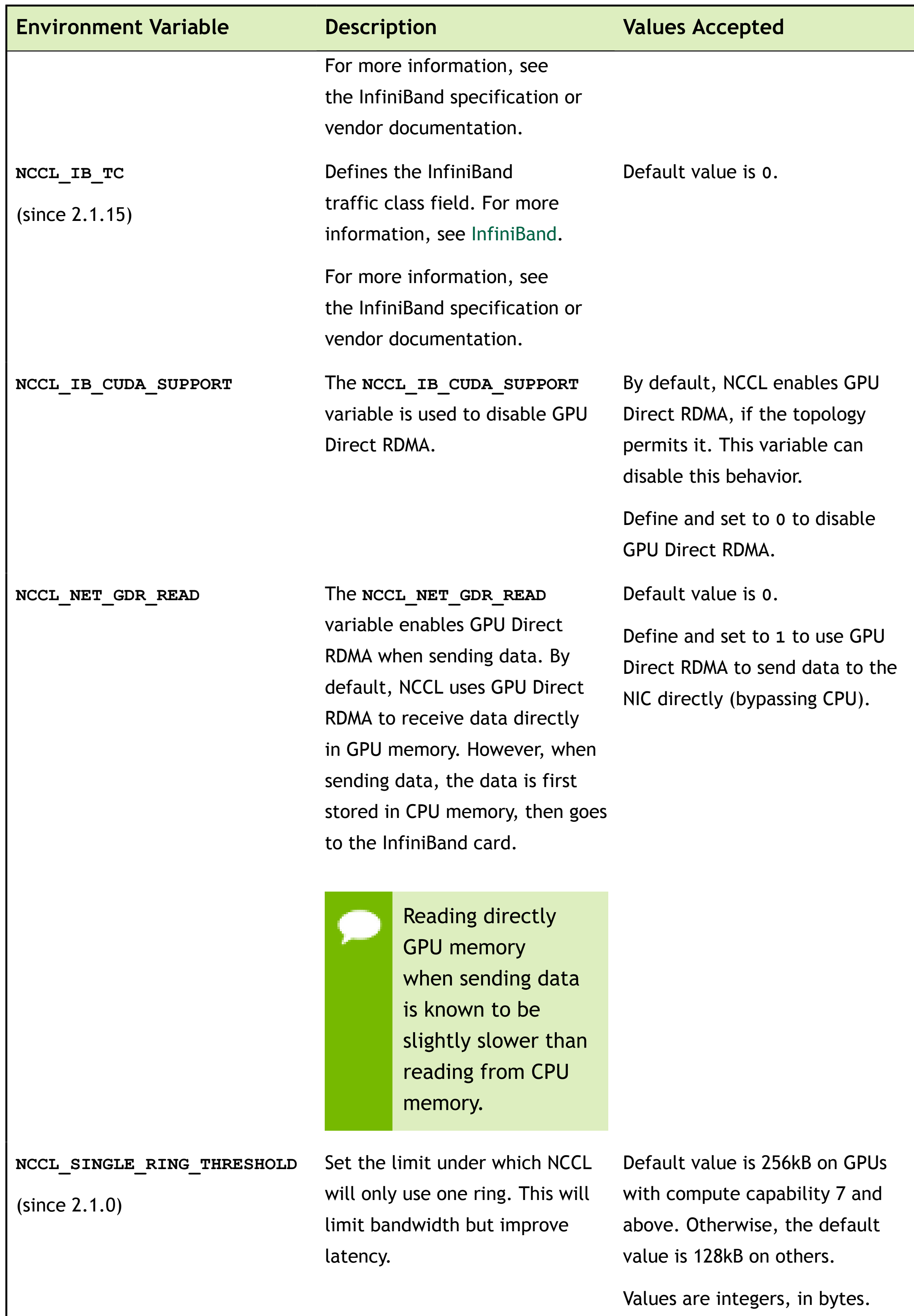


Environment Variable	Description	Values Accepted
NCCL_SHM_DISABLE	The **NCCL_SHM_DISABLE** variable disables the Shared Memory (SHM) transports. SHM is used between devices when peer-to-peer cannot happen, therefore, host memory is used. NCCL uses network (InfiniBand or IP sockets) to communicate between 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 the environment variable.
NCCL_DEBUG	The **NCCL_DEBUG** variable controls the debug information that is displayed from NCCL.	**VERSION** Prints the NCCL version at the start of the program.

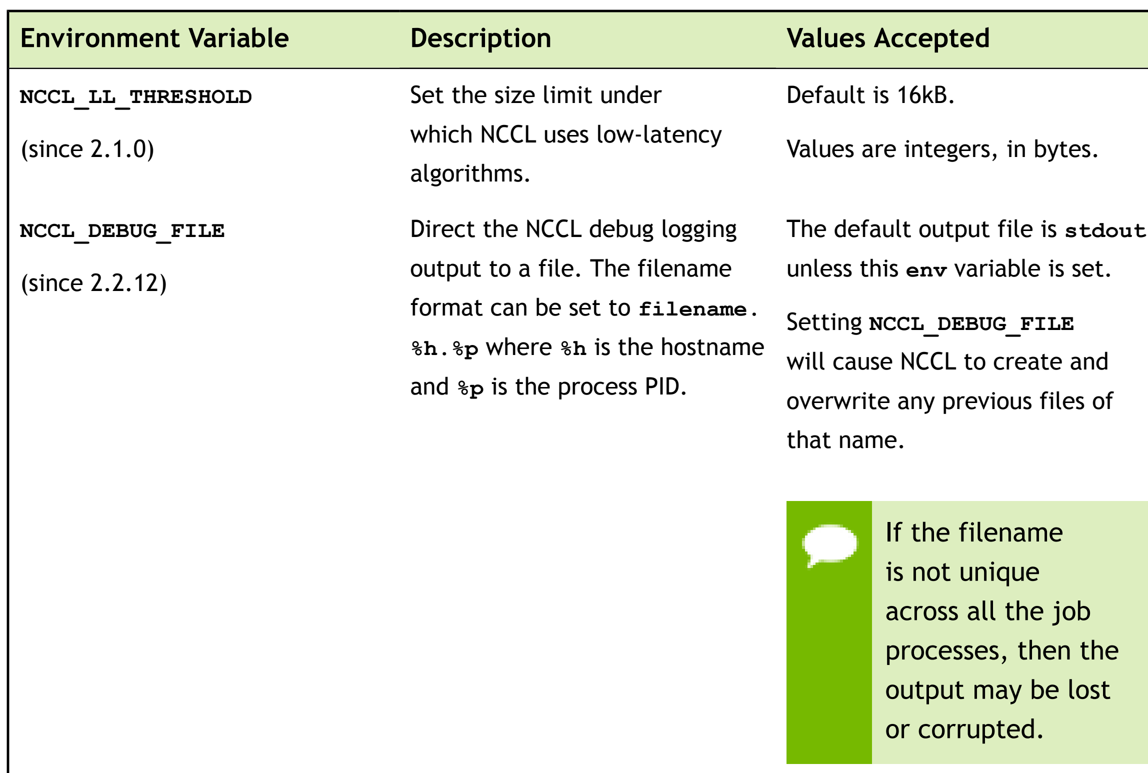
Environment Variable	Description	Values Accepted
	This variable is commonly used for debugging.	**WARN** Prints an explicit error message whenever any NCCL call errors out.
NCCL_IB_DISABLE	The **NCCL_IB_DISABLE** variable disables the IB transport that is to be used by NCCL. Instead, NCCL will fallback 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 **256**. The values allowed are **64**, **128** and **256**.
NCCL_RINGS	The **NCCL_RINGS** variable overrides the rings that NCCL forms by default. Rings are 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	Ranks from **0** to **n-1**, where **n** is the number of GPUs in your communicator. The ranks can be separated by any non-digit character, for example, " ", "-", except "|". Multiple rings can be specified separated by the pipe character "|".

Environment Variable	Description	Values Accepted
	topology connecting the GPUs in your system.	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_MAX_NRINGS (since 2.0.5)	The **NCCL_MAX_NRINGS** variable limits the number of rings NCCL can use. Reducing the number of rings also reduces the number of CUDA blocks used for communication, hence the impact on GPU computing resources.	Any value above or equal to `1`.
NCCL_MIN_NRINGS (since 2.2.0)	Controls the minimum number of rings you want NCCL to use. Increasing the number of rings also increases the number of CUDA blocks NCCL uses, which may be useful to improve performance; however, it uses more CUDA compute resources. This is especially useful when using aggregated collectives on platforms where NCCL would usually only create one ring.	Default is platform dependent. Set to a integer value, up to `12`.
NCCL_CHECKS_DISABLE (since 2.0.5) (deprecated in 2.2.12)	Disable argument checks. Checks are useful during development but can increase the latency. They can be disabled to improve performance in production.	Default is `0`. Set the value to `1` to disable checks.
NCCL_CHECK_POINTERS (since 2.2.12)	Enable checking of the CUDA memory pointers on each collective call. Checks are useful during development but can increase the latency.	Default is `0`, set to `1` to enable checking. Setting to `1` restores the original behavior of NCCL prior to 2.2.12.
NCCL_LAUNCH_MODE (since 2.1.0)	Controls how NCCL launches CUDA kernels.	The default value is to use cooperative groups (CUDA 9).

Environment Variable	Description	Values Accepted
		Setting it to **PARALLEL** uses the previous launch system which can be faster but is prone to deadlocks.
NCCL_IB_TIMEOUT	The **NCCL_IB_TIMEOUT** variable controls the InfiniBand Verbs Timeout. For more information, see InfiniBand. The timeout is computed as **4.096 µs * 2 ^** timeout, and the right value is dependent on the size of the network. Increasing that value can help on very large networks, for example, if NCCL is failing on a call to **ibv_poll_cq** with error **12**. For more information, see section 12.7.34 of the InfiniBand specification (Local Ack Timeout).	The default value used by NCCL is **14**. Values can be 1-22.
NCCL_IB_RETRY_CNT(since 2.1.15)	Controls the InfiniBand retry count. For more information, see InfiniBand. For more information, see section 12.7.38 of the InfiniBand specification.	Default value is **7**.
NCCL_IB_GID_INDEX (since 2.1.4)	Defines the Global ID index used in RoCE mode. See the **show_gids** command to set this value. For more information, see InfiniBand. For more information, see the InfiniBand specification or vendor documentation.	Default value is **0**.
NCCL_IB_SL (since 2.1.4)	Defines the InfiniBand Service Level. For more information, see InfiniBand.	Default value is **1**.



Environment Variable	Description	Values Accepted
	For more information, see the InfiniBand specification or vendor documentation.	
NCCL_IB_TC (since 2.1.15)	Defines the InfiniBand traffic class field. For more information, see InfiniBand. For more information, see the InfiniBand specification or vendor documentation.	Default value is `0`.
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. Note: Reading directly GPU memory when sending data is known to be slightly slower than reading from CPU memory.	Default value is `0`. Define and set to `1` to use GPU Direct RDMA to send data to the NIC directly (bypassing CPU).
NCCL_SINGLE_RING_THRESHOLD (since 2.1.0)	Set the limit under which NCCL will only use one ring. This will limit bandwidth but improve latency.	Default value is 256kB on GPUs with compute capability 7 and above. Otherwise, the default value is 128kB on others. Values are integers, in bytes.



Environment Variable	Description	Values Accepted
NCCL_LL_THRESHOLD (since 2.1.0)	Set the size limit under which NCCL uses low-latency algorithms.	Default is 16kB. Values are integers, in bytes.
NCCL_DEBUG_FILE (since 2.2.12)	Direct the NCCL debug logging output to a file. The filename format can be set to **filename.%h.%p** where **%h** is the hostname and **%p** is the process PID.	The default output file is **stdout** unless this **env** variable is set. Setting **NCCL_DEBUG_FILE** will cause NCCL to create and overwrite any previous files of that name. If the filename is not unique across all the job processes, then the output may be lost or corrupted.

5.5. 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.

Notice

THE INFORMATION IN THIS GUIDE AND ALL OTHER INFORMATION CONTAINED IN NVIDIA DOCUMENTATION REFERENCED IN THIS GUIDE IS PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE INFORMATION FOR THE PRODUCT, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the product described in this guide shall be limited in accordance with the NVIDIA terms and conditions of sale for the product.

THE NVIDIA PRODUCT DESCRIBED IN THIS GUIDE IS NOT FAULT TOLERANT AND IS NOT DESIGNED, MANUFACTURED OR INTENDED FOR USE IN CONNECTION WITH THE DESIGN, CONSTRUCTION, MAINTENANCE, AND/OR OPERATION OF ANY SYSTEM WHERE THE USE OR A FAILURE OF SUCH SYSTEM COULD RESULT IN A SITUATION THAT THREATENS THE SAFETY OF HUMAN LIFE OR SEVERE PHYSICAL HARM OR PROPERTY DAMAGE (INCLUDING, FOR EXAMPLE, USE IN CONNECTION WITH ANY NUCLEAR, AVIONICS, LIFE SUPPORT OR OTHER LIFE CRITICAL APPLICATION). NVIDIA EXPRESSLY DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS FOR SUCH HIGH RISK USES. NVIDIA SHALL NOT BE LIABLE TO CUSTOMER OR ANY THIRD PARTY, IN WHOLE OR IN PART, FOR ANY CLAIMS OR DAMAGES ARISING FROM SUCH HIGH RISK USES.

NVIDIA makes no representation or warranty that the product described in this guide will be suitable for any specified use without further testing or modification. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer's sole responsibility to ensure the product is suitable and fit for the application planned by customer and to do the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer's product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/ or requirements beyond those contained in this guide. NVIDIA does not accept any liability related to any default, damage, costs or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this guide, or (ii) customer product designs.

Other than the right for customer to use the information in this guide with the product, no other license, either expressed or implied, is hereby granted by NVIDIA under this guide. Reproduction of information in this guide is permissible only if reproduction is approved by NVIDIA in writing, is reproduced without alteration, and is accompanied by all associated conditions, limitations, and notices.

Trademarks

NVIDIA, the NVIDIA logo, and cuBLAS, CUDA, cuDNN, cuFFT, cuSPARSE, DIGITS, DGX, DGX-1, Jetson, Kepler, NVIDIA Maxwell, NCCL, NVLink, Pascal, Tegra, TensorRT, and Tesla are trademarks and/or registered trademarks of NVIDIA Corporation in the Unites States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2018 NVIDIA Corporation. All rights reserved.