



NVIDIA COLLECTIVE COMMUNICATION LIBRARY (NCCL)

PR-08594-001_v | September 2018

API



TABLE OF CONTENTS

Chapter 1. NCCL API.....	1
1.1. Communicator Creation And Management Functions.....	1
1.2. Collective Communication Functions.....	4
1.2.5. ncclReduceScatter.....	7
1.3. Group Calls.....	8
1.4. Types.....	8
1.5. Constants.....	10

Chapter 1.

NCCL API

The following sections describe the collective communications methods and operations.

1.1. Communicator Creation And Management Functions

The following functions are public APIs exposed by NVIDIA® Collective Communications Library™ (NCCL) to create and manage the collective communication operations.

1.1.1. `ncclGetVersion`

The `ncclGetVersion` function returns the version number of the currently linked NCCL library. The NCCL version number is encoded as an integer which includes the `NCCL_MAJOR`, `NCCL_MINOR` and `NCCL_PATCH` levels. The version number returned will be the same as the `NCCL_VERSION_CODE` defined in `nccl.h`. NCCL version numbers can be compared using the supplied macro; `NCCL_VERSION(MAJOR, MINOR, PATCH)`.

```
ncclResult_t ncclGetVersion(int* version);
```

The following table lists the arguments that are passed to the `ncclGetVersion` function.

Type	Argument Name	Description
<code>int*</code>	<code>version</code>	Pointer to an integer to be used to return the NCCL version number.

1.1.2. `ncclGetUniqueId`

The `ncclGetUniqueId` function generates an `Id` to be used in the `ncclCommInitRank` function.

The `ncclGetUniqueId` function should be called once. The `Id` should be distributed to all of the ranks in the communicator before calling the `ncclCommInitRank` function.

```
ncclResult_t ncclGetUniqueId(ncclUniqueId* uniqueId);
```

The following table lists the arguments that are passed to the `ncclGetUniqueId` function.

Type	Argument Name	Description
<code>ncclUniqueId*</code>	<code>uniqueId</code>	Pointer to an already allocated unique <code>Id</code> .

1.1.3. `ncclCommInitRank`

The `ncclCommInitRank` function creates a new communicator object for the current CUDA[®] device. This function allows for multi-process initialization.

```
ncclResult_t ncclCommInitRank(ncclComm_t* comm, int nranks, ncclUniqueId
commId, int
rank);
```

The `ncclCommInitRank` function implicitly synchronizes with other ranks, so it must be called by different threads and processes or use the `ncclGroupStart` and `ncclGroupEnd` functions.

The following table lists the arguments that are passed to the `ncclCommInitRank` function.

Type	Argument Name	Description
<code>ncclComm_t*</code>	<code>comm</code>	Returned communicator.
<code>int</code>	<code>nranks</code>	Number of ranks in the communicator.
<code>ncclUniqueId*</code>	<code>uniqueId</code>	Pointer to a unique <code>Id</code> .
<code>int</code>	<code>rank</code>	The rank associated to the current device. The rank must be between 0 and <code>nranks-1</code> and unique within the communicator clique.

1.1.4. `ncclCommInitAll`

The `ncclCommInitAll` function creates a full communicator. For example, a clique of communicator objects. The communicator only works within a single process.

```
ncclResult_t ncclCommInitAll(ncclComm_t* comm, int ndev, const int* devlist);
```

The `ncclCommInitAll` function returns an array of **`ndev`** newly initialized communicators in **`comm`**. The argument name **`comm`**, should be pre-allocated with the

size of at least `ndev*sizeof\(ncclComm_t\)`. If `devlist` is `NULL`, the first `ndev` CUDA devices are used. The order of `devlist` defines the user order of the devices within the communicator.

The following table lists the arguments that are passed to the `ncclCommInitAll` function.

Type	Argument Name	Description
<code>ncclComm_t*</code>	<code>comm</code>	Returned array of communicators. The <code>comm</code> argument should be pre-allocated with a size of at least: <code>ndev*sizeof(ncclComm_t)</code> .
<code>int</code>	<code>ndev</code>	Number of ranks or devices in the communicator.
<code>const int*</code>	<code>devlist</code>	A list of CUDA devices to associate with each rank. Should be an array of <code>ndev</code> integers.

1.1.5. `ncclCommDestroy`

The `ncclCommDestroy` function frees resources that are allocated to a communicator object.

```
ncclResult_t ncclCommDestroy(ncclComm_t comm);
```

The following table lists the arguments that are passed to the `ncclCommDestroy` function.

Type	Argument Name	Description
<code>ncclComm_t</code>	<code>comm</code>	Communicator object to free.

1.1.6. `ncclCommCount`

The `ncclCommCount` function returns the number of ranks in a communicator.

```
ncclResult_t ncclCommCount(const ncclComm_t comm, int* count);
```

The following table lists the arguments that are passed to the `ncclCommCount` function.

Type	Argument Name	Description
<code>ncclComm_t</code>	<code>comm</code>	Communicator object.
<code>int*</code>	<code>count</code>	Number of ranks returned.

1.1.7. `ncclCommCuDevice`

The `ncclCommCuDevice` function returns the CUDA device associated with a communicator object.

```
ncclResult_t ncclCommCuDevice(const ncclComm_t comm, int* device);
```

The following table lists the arguments that are passed to the `ncclCommCuDevice` function.

Type	Argument Name	Description
<code>ncclComm_t</code>	<code>comm</code>	Communicator object.
<code>int*</code>	<code>count</code>	CUDA device returned.

1.1.8. `ncclCommUserRank`

The `ncclCommUserRank` function returns the rank of a communicator object.

```
ncclResult_t ncclCommUserRank(const ncclComm_t comm, int* rank);
```

The following table lists the arguments that are passed to the `ncclCommUserRank` function.

Type	Argument Name	Description
<code>ncclComm_t</code>	<code>comm</code>	Communicator object.
<code>int*</code>	<code>rank</code>	Rank returned.

1.2. Collective Communication Functions

The following NCCL APIs provide some commonly used collective operations.

1.2.1. `ncclAllReduce`

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

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

The following table lists the arguments that are passed to the `ncclAllReduce` function.

Type	Argument Name	Description
<code>const void*</code>	<code>sendbuff</code>	Pointer to the data to read from.

Type	Argument Name	Description
void*	recvbuff	Pointer to the data to write to.
size_t	count	Number of elements to process.
ncclDataType_t	datatype	Type of element.
ncclRedOp_t	op	Operation to perform on each element.
ncclComm_t	comm	Communicator object.
cudaStream_t	stream	CUDA stream to run the operation on.

1.2.2. ncclBroadcast

The `ncclBroadcast` function copies the count values from the root rank to all ranks.

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

The `ncclBcast` function is a legacy in-place version of `ncclBroadcast` in a similar fashion to `MPI_Bcast`. A call to `ncclBcast (buff, count, datatype, root, comm, stream)` is equivalent to `ncclBroadcast (buff, count, datatype, root, comm, stream)`.

```
ncclResult_t ncclBcast(void* buff, size_t count, ncclDataType_t datatype, int
    root, ncclComm_t comm, cudaStream_t stream);
```

The following table lists the arguments that are passed to the `ncclBroadcast` function.

Type	Argument Name	Description
const void*	sendbuff	Pointer to the data to read from.
void*	recvbuff	Pointer to the data to read to.
size_t	count	Number of elements to process.
ncclDataType_t	datatype	Type of element.
int	root	Rank of the root of the operation.
ncclComm_t	comm	Communicator object.
cudaStream_t	stream	CUDA stream to run the operation on.

1.2.3. `ncclReduce`

The `ncclReduce` function reduces data arrays of length `count` in `sendbuff` into `recvbuff` using the `op` operation.

```
ncclResult_t ncclReduce(const void* sendbuff, void* recvbuff, size_t count,
                        ncclDataType_t datatype,
                        ncclRedOp_t op, int root, ncclComm_t comm, cudaStream_t stream);
```

The following table lists the arguments that are passed to the `ncclReduce` function.

Type	Argument Name	Description
<code>const void*</code>	<code>sendbuff</code>	Pointer to the data to read from.
<code>void*</code>	<code>recvbuff</code>	Pointer to the data to write to.
<code>size_t</code>	<code>count</code>	Number of elements to process.
<code>ncclDataType_t</code>	<code>datatype</code>	Type of element.
<code>ncclRedOp_t</code>	<code>op</code>	Operation to perform on each element.
<code>int</code>	<code>root</code>	Rank of the root of the operation.
<code>ncclComm_t</code>	<code>comm</code>	Communicator object.
<code>cudaStream_t</code>	<code>stream</code>	CUDA stream to run the operation on.

1.2.4. `ncclAllGather`

The `ncclAllGather` function gathers `sendcount` values from other GPUs into `recvbuff`, receiving data from rank `i` at offset `i*sendcount`.



This assumes `recvcount` is equal to `n ranks * sendcount`, which means that `recvbuff` should have a size of at least `n ranks * sendcount` elements.

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


The following table lists the arguments that are passed to the `ncclAllGather` function.

Type	Argument Name	Description
<code>const void*</code>	<code>sendbuff</code>	Pointer to the data to read from.

Type	Argument Name	Description
void*	recvbuff	Pointer to the data to write to. This should be the size of sendcount*nranks .
size_t	sendcount	Number of elements sent per rank.
ncclDataType_t	datatype	Type of element.
int	root	Rank of the root of the operation.
ncclComm_t	comm	Communicator object.
cudaStream_t	stream	CUDA stream to run the operation on.

1.2.5. ncclReduceScatter

The `ncclReduceScatter` function reduces data in `sendbuff` using the `op` operation and leaves the reduced result scattered over the devices so that the `recvbuff` on rank `i` will contain the `i-th` block of the result.

 This assumes `sendcount` is equal to `nranks*recvcount`, which means that `sendbuff` should have a size of at least `nranks*recvcount` elements.

```
ncclResult_t ncclReduceScatter(const void* sendbuff, void* recvbuff,
                               size_t recvcount, ncclDataType_t datatype, ncclRedOp_t op, ncclComm_t comm,
                               cudaStream_t stream);
```

The following table lists the arguments that are passed to the `ncclReduceScatter` function.

Type	Argument Name	Description
const void*	sendbuff	Pointer to the data to read from. This should be the size of recvcount*nranks .
void*	recvbuff	Pointer to the data to write to.
size_t	recvcount	Number of elements to receive by each rank.
ncclDataType_t	datatype	Type of element.
ncclRedOp_t	op	Operation to perform on each element.
ncclComm_t	comm	Communicator object.

Type	Argument Name	Description
<code>cudaStream_t</code>	<code>stream</code>	CUDA stream to run the operation on.

1.3. Group Calls

Group primitives define the behavior of the current thread to avoid blocking. They can therefore be used from multiple threads independently.

1.3.1. `ncclGroupStart`

The `ncclGroupStart` call starts a group call.

All subsequent calls to NCCL may not block due to inter-CPU synchronization.

```
ncclResult_t ncclGroupStart();
```

1.3.2. `ncclGroupEnd`

The `ncclGroupEnd` call ends a group call.

The `ncclGroupEnd` call returns when all operations since `ncclGroupStart` have been processed. This means communication primitives have been enqueued to the provided streams, but are not necessary complete. When used with `ncclCommInitRank`, it means all communicators have been initialized and are ready to be used.

When the `ncclGroupEnd` call is used with the `ncclCommInitRank` function, the `ncclGroupEnd` call waits for all communicators to be initialized.

```
ncclResult_t ncclGroupEnd();
```

1.4. Types

The following types are used by the CUDA library. These types are useful when configuring your collective operations.

1.4.1. `ncclDataType_t`

NCCL defines the following integral and floating data-types.

Data-Type	Description
<code>ncclInt8</code> , <code>ncclChar</code>	Signed 8-bits integer.
<code>ncclUInt8</code>	Unsigned 8-bits integer.
<code>ncclInt32</code> , <code>ncclInt</code>	Signed 32-bits integer.
<code>ncclUInt32</code>	Unsigned 32-bits integer.

Data-Type	Description
<code>ncclInt64</code>	Signed 64-bits integer.
<code>ncclUInt64</code>	Unsigned 64-bits integer.
<code>ncclFloat16</code> , <code>ncclHalf</code>	16-bits floating point number (half precision)
<code>ncclFloat32</code> , <code>ncclFloat</code>	32-bits floating point number (single precision)
<code>ncclFloat64</code> , <code>ncclDouble</code>	64-bits floating point number (double precision)

1.4.2. `ncclRedOp_t`

NCCL defines the following reduction operations.

Reduction Operation	Description
<code>ncclSum</code>	Perform a sum (+) operation.
<code>ncclProd</code>	Perform a product (*) operation.
<code>ncclMin</code>	Perform a min operation.
<code>ncclMax</code>	Perform a max operation.

1.4.3. `ncclResult_t`

NCCL functions always return an error code of type `ncclResult_t`.

If the `NCCL_DEBUG` environment variable is set to **WARN**, whenever a function returns an error, NCCL should print the reason.

Return Code	Description
<code>ncclSuccess</code>	The operations completed successfully.
<code>ncclUnhandledCudaError</code>	A call to CUDA returned a fatal error for the NCCL operation.
<code>ncclSystemError</code>	A call to the system returned a fatal error for the NCCL operation.
<code>ncclInternalError</code>	NCCL experienced an internal error.
<code>ncclInvalidArgument</code>	The user has supplied an invalid argument.
<code>ncclInvalidUsage</code>	The user has used NCCL in an invalid manner.

1.5. Constants

NCCL defines two constants `NCCL_MAJOR` and `NCCL_MINOR` to help distinguish between API changes, in particular between NCCL 1.x and NCCL 2.x.

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, DALI, 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 United 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.