RCCL Documentation

Release 2.27.7

Advanced Micro Devices, Inc.

May 05, 2026

CONTENTS

The ROCm Communication Collectives Library (RCCL) is a stand-alone library that provides multi-GPU and multi-node collective communication primitives optimized for AMD GPUs. It uses PCIe and xGMI high-speed interconnects. To learn more, see *What is RCCL?*

The RCCL public repository is located at https://github.com/ROCm/rccl.

Install

- *Installing RCCL using the install script*
- *Running RCCL using Docker*
- *Building and installing RCCL from source code*

How to

- *Using the RCCL Tuner plugin*
- *Using the NCCL Net plugin*
- *Troubleshoot RCCL*
- *RCCL usage tips*

Examples

- RCCL Tuner plugin examples
- NCCL Net plugin examples

API reference

- *Library specification*
- *API library*
- *Environment variables*

To contribute to the documentation, see Contributing to ROCm.

You can find licensing information on the Licensing page.

CHAPTER

ONE

WHAT IS RCCL?

The ROCm Communication Collectives Library (RCCL) includes multi-GPU and multi-node collective communication primitives optimized for AMD GPUs. It implements routines such as `all-reduce`, `all-gather`, `reduce`, `broadcast`, `reduce-scatter`, `gather`, `scatter`, `all-to-allv`, and `all-to-all`, as well as direct point-to-point (GPU-to-GPU) send and receive operations. It is optimized to achieve high bandwidth on platforms using PCIe and xGMI and networking using InfiniBand Verbs or TCP/IP sockets. RCCL supports an arbitrary number of GPUs installed in a single node or multiple nodes and can be used in either single- or multi-process (for example, MPI) applications.

The collective operations are implemented using ring and tree algorithms and have been optimized for throughput and latency by leveraging topology awareness, high-speed interconnects, and RDMA-based collectives. For best performance, small operations can be either batched into larger operations or aggregated through the API.

RCCL uses PCIe and xGMI high-speed interconnects for intra-node communication as well as InfiniBand, RoCE, and TCP/IP for inter-node communication. It supports an arbitrary number of GPUs installed in a single-node or multi-node platform and can easily integrate into single- or multi-process (for example, MPI) applications.

CHAPTER

TWO

INSTALLING RCCL USING THE INSTALL SCRIPT

To quickly install RCCL using the install script, follow these steps. For instructions on building RCCL from the source code, see *Building and installing RCCL from source code*. For additional tips, see *RCCL usage tips*.

2.1 Requirements

The following prerequisites are required to use RCCL:

1. ROCm-supported GPUs
2. The ROCm stack must be installed on the system, including the HIP runtime and the HIP-Clang compiler.

2.2 Quick start RCCL build

RCCL directly depends on the HIP runtime plus the HIP-Clang compiler, which are part of the ROCm software stack. For ROCm installation instructions, see the package manager installation guide.

Use the install.sh helper script, located in the root directory of the RCCL repository, to build and install RCCL with a single command. It uses hard-coded configurations that can be specified directly when using cmake. However, it's a great way to get started quickly and provides an example of how to build and install RCCL.

2.2.1 Building the library using the install script:

To build the library using the install script, use this command:

```
./install.sh
```

For more information on the build options and flags for the install script, run the following command:

```
./install.sh --help
```

The RCCL build and installation helper script options are as follows:

```
   --address-sanitizer     Build with address sanitizer enabled
-d|--dependencies          Install RCCL dependencies
   --debug                 Build debug library
   --enable_backtrace      Build with custom backtrace support
   --disable-colltrace     Build without collective trace
   --disable-msccl-kernel  Build without MSCCL kernels
```

(continues on next page)

(continued from previous page)

```
   --enable-mscclpp        Build with MSCCL++ support
-f|--fast                  Quick-build RCCL (local gpu arch only, no backtrace, and␣
↪collective trace support)
-h|--help                  Prints this help message
-i|--install               Install RCCL library (see --prefix argument below)
-j|--jobs                  Specify how many parallel compilation jobs to run ($nproc by␣
↪default)
-l|--local_gpu_only        Only compile for local GPU architecture
   --amdgpu_targets        Only compile for specified GPU architecture(s). For multiple␣
↪targets, separate by ';' (builds for all supported GPU architectures by default)
   --no_clean              Don't delete files if they already exist
   --npkit-enable          Compile with npkit enabled
   --openmp-test-enable    Enable OpenMP in rccl unit tests
   --roctx-enable          Compile with roctx enabled (example usage: rocprof --roctx-
↪trace ./rccl-program)
-p|--package_build         Build RCCL package
   --prefix                Specify custom directory to install RCCL to (default: `/opt/
↪rocm`)
   --rm-legacy-include-dir Remove legacy include dir Packaging added for file/folder␣
↪reorg backward compatibility
   --run_tests_all         Run all rccl unit tests (must be built already)
-r|--run_tests_quick       Run small subset of rccl unit tests (must be built already)
   --static                Build RCCL as a static library instead of shared library
-t|--tests_build           Build rccl unit tests, but do not run
   --time-trace            Plot the build time of RCCL (requires `ninja-build` package␣
↪installed on the system)
   --verbose               Show compile commands
```

Tip: By default, the RCCL install script builds all the GPU targets that are defined in `DEFAULT_GPUS` in CMakeLists.txt. To target specific GPUs and potentially reduce the build time, use `--amdgpu_targets` along with a semicolon (`;`) separated string list of the GPU targets.

CHAPTER

THREE

RUNNING RCCL USING DOCKER

To use Docker to run RCCL, Docker must already be installed on the system. To build the Docker image and run the container, follow these steps.

1. Build the Docker image

   By default, the Dockerfile uses `docker.io/rocm/dev-ubuntu-22.04:latest` as the base Docker image. It then installs RCCL and rccl-tests (in both cases, it uses the version from the `develop` branch).

   Use this command to build the Docker image:

   ```
   docker build -t rccl-tests -f Dockerfile.ubuntu --pull .
   ```

   The base Docker image, rccl repository, rccl-tests repository, and GPU targets can be modified by using `--build-args` in the `docker build` command above. For example, to use a different base Docker image for the MI250 GPU, use this command:

   ```
   docker build -t rccl-tests -f Dockerfile.ubuntu --build-arg="ROCM_IMAGE_NAME=rocm/
   ↪dev-ubuntu-20.04" --build-arg="ROCM_IMAGE_TAG=6.2" --build-arg="GPU_TARGETS=gfx90a
   ↪" --pull .
   ```

2. Launch an interactive Docker container on a system with AMD GPUs:

   ```
   docker run --rm --device=/dev/kfd --device=/dev/dri --group-add video --ipc=host --
   ↪network=host --cap-add=SYS_PTRACE --security-opt seccomp=unconfined -it rccl-
   ↪tests /bin/bash
   ```

To run, for example, the `all_reduce_perf` test from rccl-tests on 8 AMD GPUs from inside the Docker container, use this command for ROCm 6.4.1 or earlier:

```
mpirun --allow-run-as-root -np 8 --mca pml ucx --mca btl ^openib -x NCCL_DEBUG=VERSION -
↪x HSA_NO_SCRATCH_RECLAIM=1 /workspace/rccl-tests/build/all_reduce_perf -b 1 -e 16G -f␣
↪2 -g 1
```

For ROCm 6.4.2 or later, use this command:

```
mpirun --allow-run-as-root -np 8 --mca pml ucx --mca btl ^openib -x NCCL_DEBUG=VERSION /
↪workspace/rccl-tests/build/all_reduce_perf -b 1 -e 16G -f 2 -g 1
```

For more information on the rccl-tests options, see the Usage guidelines in the GitHub repository.

CHAPTER

FOUR

BUILDING AND INSTALLING RCCL FROM SOURCE CODE

To build RCCL directly from the source code, follow these steps. This guide also includes instructions explaining how to test the build. For information on using the quick start install script to build RCCL, see *Installing RCCL using the install script*.

4.1 Requirements

The following prerequisites are required to build RCCL:

1. ROCm-supported GPUs
2. Having the ROCm stack installed on the system, including the HIP runtime and the HIP-Clang compiler.

4.1.1 Building the library using CMake:

To build the library from source, follow these steps:

```
git clone --recursive https://github.com/ROCm/rccl.git
cd rccl
mkdir build
cd build
cmake ..
make -j 16      # Or some other suitable number of parallel jobs
```

If you have already cloned the repository, you can checkout the external submodules manually.

```
git submodule update --init --recursive --depth=1
```

You can substitute a different installation path by providing the path as a parameter to `CMAKE_INSTALL_PREFIX`, for example:

```
cmake -DCMAKE_INSTALL_PREFIX=$PWD/rccl-install -DCMAKE_BUILD_TYPE=Release ..
```

Note: Ensure ROCm CMake is installed using the command `apt install rocm-cmake`. By default, CMake builds the component in debug mode unless `DCMAKE_BUILD_TYPE` is specified.

4.1.2 Building the RCCL package and install package:

After you have cloned the repository and built the library as described in the previous section, use this command to build the package:

```
cd rccl/build
make package
sudo dpkg -i *.deb
```

Note: The RCCL package install process requires `sudo` or root access because it creates a directory named `rccl` in `/opt/rocm/`. This is an optional step. RCCL can be used directly by including the path containing `librccl.so`.

4.2 Testing RCCL

The RCCL unit tests are implemented using the Googletest framework in RCCL. These unit tests require Googletest 1.10 or higher to build and run (this dependency can be installed using the `-d` option for `install.sh`). To run the RCCL unit tests, go to the `build` folder and the `test` subfolder, then run the appropriate RCCL unit test executables.

The RCCL unit test names follow this format:

```
CollectiveCall.[Type of test]
```

Filtering of the RCCL unit tests can be done using environment variables and by passing the `--gtest_filter` command line flag:

```
UT_DATATYPES=ncclBfloat16 UT_REDOPS=prod ./rccl-UnitTests --gtest_filter="AllReduce.C*"
```

This command runs only the `AllReduce` correctness tests with the `float16` datatype. A list of the available environment variables for filtering appears at the top of every run. See the Googletest documentation for more information on how to form advanced filters.

There are also other performance and error-checking tests for RCCL. They are maintained separately at https://github.com/ROCm/rccl-tests.

Note: For more information on how to build and run rccl-tests, see the rccl-tests README file .

CHAPTER

FIVE

USING THE RCCL TUNER PLUGIN API

An external plugin enables users to hand-tailor the selection of an algorithm, protocol, and number of channels (thread blocks) based on an input configuration specifying the message size, number of nodes and GPUs, and link types (for instance, PCIe, XGMI, or NET). One advantage of this plugin is that each user can create and maintain their own hand-tailored tuner without relying on RCCL to develop and maintain it. This topic describes the API required to implement an external tuner plugin for RCCL.

The following usage notes are relevant when using the RCCL Tuner plugin API:

- The API allows partial outputs: tuners can set only the algorithm and protocol and let RCCL set the remaining fields, such as the number of channels.
- If `getCollInfo()` fails, RCCL uses its default internal mechanisms to determine the best collective configuration.
- `getCollInfo` is called for each collective invocation per communicator, so special care must be taken to avoid introducing excessive latency.
- The supported RCCL algorithms are `NCCL_ALGO_TREE`, and `NCCL_ALGO_RING`.
- The supported RCCL protocols are `NCCL_PROTO_SIMPLE`, `NCCL_PROTO_LL` and `NCCL_PROTO_LL128`.
  - Until support is present for network collectives, use the example in the `pluginGetCollInfo` API implementation to ignore other algorithms as follows:

```
if ((a == NCCL_ALGO_COLLNET_DIRECT || a == NCCL_ALGO_COLLNET_CHAIN) &&␣
↪collNetSupport != 1) continue;
if ((a == NCCL_ALGO_NVLS || a == NCCL_ALGO_NVLS_TREE) && nvlsSupport != 1)␣
↪continue;
if (a == NCCL_ALGO_NVLS && collNetSupport != 1) continue;
```

Note: The example plugin uses math models to approximate the bandwidth and latency of the available selection of algorithms and protocols and select the one with the lowest calculated latency. It is customized for the AMD Instinct MI300 accelerators and RoCEv2 networks on a limited number of nodes. This example, which is intended for demonstration purposes only, is not meant to be inclusive of all potential AMD GPUs and network configuration.

5.1 API description

To build a custom tuner, implement the `ncclTuner_v1_t` structure.

5.1.1 Structure: ncclTuner_v1_t

Fields

- `name`
  - **Type**: `const char*`
  - **Description**: The name of the tuner, which can be used for logging purposes when `NCCL_DEBUG=info` and `NCCL_DEBUG_SUBSYS=tune` are set.

Functions

- `init` (called upon communicator initialization with `ncclCommInitRank`)

  Initializes the tuner states. Each communicator initializes its tuner. `nNodes` x `nRanks` = the total number of GPUs participating in the collective communication.
  - **Parameters**:
    - `nRanks` (`size_t`): The number of devices (GPUs).
    - `nNodes` (`size_t`): The number of operating system nodes (physical nodes or VMs).
    - `logFunction` (`ncclDebugLogger_t`): A log function for certain debugging info.
  - **Return**:
    - **Type**: `ncclResult_t`
    - **Description**: The result of the initialization.
- `getCollInfo` (called for each collective call per communicator)

  Retrieves information about the collective algorithm, protocol, and number of channels for the given input parameters.
  - **Parameters**:
    - `collType` (`ncclFunc_t`): The collective type, for example, `allreduce`, `allgather`, etc.
    - `nBytes` (`size_t`): The size of the collective in bytes.
    - `collNetSupport` (`int`): Whether `collNet` supports this type.
    - `nvlsSupport` (`int`): Whether NVLink SHARP supports this type.
    - `numPipeOps` (`int`): The number of operations in the group.
  - **Outputs**:
    - `algorithm` (`int*`): The selected algorithm to be used for the given collective.
    - `protocol` (`int*`): The selected protocol to be used for the given collective.
    - `nChannels` (`int*`): The number of channels (and SMs) to be used.
  - **Description**:

    If `getCollInfo()` does not return `ncclSuccess`, RCCL falls back to its default tuning for the given collective. The tuner is allowed to leave fields unset, in which case RCCL automatically sets those fields.

  - **Return**:
    - **Type**: `ncclResult_t`
    - **Description**: The result of the operation.

- `destroy` (called upon communicator finalization with `ncclCommFinalize`)

  Terminates the plugin and cleans up any resources allocated by the tuner.

  - **Return**:
    - **Type**: `ncclResult_t`
    - **Description**: The result of the cleanup process.

5.2 Build and usage instructions

To use the external plugin, implement the desired algorithm and protocol selection technique using the API described above. As a reference, the following example is based on the MI300 tuning table by default.

5.2.1 Building and using the example libnccl-tuner.so file

1. Build the `libnccl-tuner.so` file following the example Makefile.

```
cd $RCCL_HOME/ext-tuner/example/
make
```

2. Tell RCCL to use the custom `libnccl-tuner.so` file by setting the following environment variable to the file path:

```
export NCCL_TUNER_PLUGIN=$RCCL_HOME/ext-tuner/example/libnccl-tuner.so
```

CHAPTER

SIX

USING THE NCCL NET PLUGIN API

NCCL provides a way to use external plugins to let NCCL run on many network types. This topic describes the NCCL Net plugin API and explains how to implement a network plugin for NCCL.

Plugins implement the NCCL network API and decouple NCCL binary builds, which are built against a particular version of the GPU stack (such as NVIDIA CUDA), from the network code, which is built against a particular version of the networking stack. Using this method, you can easily integrate any CUDA version with any network stack version.

NCCL network plugins are packaged as a shared library called `libnccl-net.so`. The shared library contains one or more implementations of the NCCL Net API in the form of versioned structs, which are filled with pointers to all required functions.

6.1 Plugin architecture

When NCCL is initialized, it searches for a `libnccl-net.so` library and dynamically loads it, then searches for symbols inside the library.

The `NCCL_NET_PLUGIN` environment variable allows multiple plugins to coexist. If it's set, NCCL looks for a library named `libnccl-net-${NCCL_NET_PLUGIN}.so`. It is therefore recommended that you name the library according to that pattern, with a symlink pointing from `libnccl-net.so` to `libnccl-net-${NCCL_NET_PLUGIN}.so`. This lets users select the correct plugin if there are multiple plugins in the path.

6.1.1 Struct versioning

After a library is found, NCCL looks for a symbol named `ncclNet_vX`, with `X` increasing over time. This versioning pattern ensures that the plugin and the NCCL core are compatible.

Plugins are encouraged to provide a number of these symbols, implementing many versions of the NCCL Net API. This is so the same plugin can be compiled for and support a wide range of NCCL versions.

Conversely, and to ease transition, NCCL can choose to support different plugin versions. It can look for the latest `ncclNet` struct version but also search for older versions, so that older plugins still work.

6.1.2 In-network collective operations (collNet)

In addition to the `ncclNet` structure, network plugins can provide a `collNet` structure which implements any supported in-network collective operations. This is an optional structure provided by the network plugin, but its versioning is tied to the `ncclNet` structure and many functions are common between the two to ease implementation. The `collNet` structure can be used by the NCCL `collNet` algorithm to accelerate inter-node reductions in allReduce.

6.1.3 Header management

To help users effortlessly build plugins, plugins should copy the `ncclNet_vX` definitions they support to their list of internal includes. An example is shown in `ext-net/example/`, which stores all headers in the `nccl/` directory and provides thin layers to implement old versions on top of newer ones.

The `nccl/` directory is populated with `net_vX.h` files, which extract all relevant definitions from the old API versions. It also provides error codes in `err.h`.

6.2 API (v6)

Here is the main `ncclNet_v6` struct. Each function is explained in later sections.

```
typedef struct {
// Name of the network (mainly for logs)
const char* name;
// Initialize the network.
ncclResult_t (*init)(ncclDebugLogger_t logFunction);
// Return the number of adapters.
ncclResult_t (*devices)(int* ndev);
// Get various device properties.
ncclResult_t (*getProperties)(int dev, ncclNetProperties_v6_t* props);
// Create a receiving object and provide a handle to connect to it. The
// handle can be up to NCCL_NET_HANDLE_MAXSIZE bytes and will be exchanged
// between ranks to create a connection.
ncclResult_t (*listen)(int dev, void* handle, void** listenComm);
// Connect to a handle and return a sending comm object for that peer.
// This call must not block for the connection to be established, and instead
// should return successfully with sendComm == NULL with the expectation that
// it will be called again until sendComm != NULL.
ncclResult_t (*connect)(int dev, void* handle, void** sendComm);
// Finalize connection establishment after remote peer has called connect.
// This call must not block for the connection to be established, and instead
// should return successfully with recvComm == NULL with the expectation that
// it will be called again until recvComm != NULL.
ncclResult_t (*accept)(void* listenComm, void** recvComm);
// Register/Deregister memory. Comm can be either a sendComm or a recvComm.
// Type is either NCCL_PTR_HOST or NCCL_PTR_CUDA.
ncclResult_t (*regMr)(void* comm, void* data, int size, int type, void** mhandle);
/* DMA-BUF support */
ncclResult_t (*regMrDmaBuf)(void* comm, void* data, size_t size, int type, uint64_t␣
↪offset, int fd, void** mhandle);
ncclResult_t (*deregMr)(void* comm, void* mhandle);
// Asynchronous send to a peer.
```

(continues on next page)

(continued from previous page)

```
// May return request == NULL if the call cannot be performed (or would block)
ncclResult_t (*isend)(void* sendComm, void* data, int size, int tag, void* mhandle,␣
↪void** request);
// Asynchronous recv from a peer.
// May return request == NULL if the call cannot be performed (or would block)
ncclResult_t (*irecv)(void* recvComm, int n, void** data, int* sizes, int* tags, void**␣
↪mhandles, void** request);
// Perform a flush/fence to make sure all data received with NCCL_PTR_CUDA is
// visible to the GPU
ncclResult_t (*iflush)(void* recvComm, int n, void** data, int* sizes, void** mhandles,␣
↪void** request);
// Test whether a request is complete. If size is not NULL, it returns the
// number of bytes sent/received.
ncclResult_t (*test)(void* request, int* done, int* sizes);
// Close and free send/recv comm objects
ncclResult_t (*closeSend)(void* sendComm);
ncclResult_t (*closeRecv)(void* recvComm);
ncclResult_t (*closeListen)(void* listenComm);
} ncclNet_v6_t;
```

6.2.1 Error codes

All plugins functions use NCCL error codes as their return value. `ncclSuccess` should be returned upon success. Otherwise, plugins can return one of the following codes:

- `ncclSystemError` is the most common error for network plugins. It should be returned when a call to the Linux kernel or a system library fails. This typically includes all network and hardware errors.
- `ncclInternalError` is returned when the NCCL core code is using the network plugin in an incorrect way, for example, allocating more requests than it should or passing an invalid argument in API calls.
- `ncclInvalidUsage` should be returned when the error is most likely due to user error. This can include misconfiguration, but also size mismatches.
- `ncclInvalidArgument` should not typically be used by plugins because arguments should be checked by the NCCL core layer.
- `ncclUnhandledCudaError` is returned when an error is received from NVIDIA CUDA. Network plugins should not need to rely on CUDA, so this error should not be common.

6.2.2 Operational overview

NCCL first calls the `init` function, queries the number of network devices with the `devices` function, and retrieves the properties from each network device using `getProperties`.

To establish a connection between two network devices, NCCL first calls `listen` on the receiving side. It passes the returned handle to the sender side of the connection, and uses it to call `connect`. Finally, `accept` is called on the receiving side to finalize the establishment of the connection.

After the connection is established, communication is performed using the functions `isend`, `irecv`, and `test`. Prior to calling `isend` or `irecv`, NCCL calls the `regMr` function on all buffers to allow RDMA NICs to prepare the buffers. `deregMr` is used to unregister buffers.

In certain conditions, `iflush` is called after a `receive` call completes to allow the network plugin to flush data and ensure the GPU processes the newly written data.

To close the connections, NCCL calls `closeListen` to close the object returned by `listen`, `closeSend` to close the object returned by `connect`, and `closeRecv` to close the object returned by `accept`.

6.2.3 API Functions

The RCCL Tuner plugin API provides the following interface for initialization, connection management, and communications.

Initialization

- `name` - The `name` field should point to a character string with the name of the network plugin. This name is used for all logging, especially when `NCCL_DEBUG=INFO` is set.

  Note: Setting `NCCL_NET=<plugin name>` ensures a specific network implementation is used, with a matching `name`. This is not to be confused with `NCCL_NET_PLUGIN` which defines a suffix for the `librccl-net.so` library name to load.

- `init` - As soon as NCCL finds the plugin and the correct `ncclNet` symbol, it calls the `init` function. This allows the plugin to discover network devices and ensure they are usable. If the `init` function does not return `ncclSuccess`, then NCCL does not use the plugin and falls back to internal ones.

  To allow the plugin logs to seamlessly integrate into the NCCL logs, NCCL provides a logging function to `init`. This function is typically used to allow `INFO` and `WARN` macros within the plugin code by adding the following definitions:

```
#define WARN(...) logFunction(NCCL_LOG_WARN, NCCL_ALL, __FILE__, __LINE__, __VA_
↪ARGS__)
#define INFO(FLAGS, ...) logFunction(NCCL_LOG_INFO, (FLAGS), __func__, __LINE__, __
↪VA_ARGS__)
```

- `devices` - After the plugin is initialized, NCCL queries the number of devices available. This should not be zero. Otherwise, NCCL initialization will fail. If no device is present or usable, the `init` function should not return `ncclSuccess`.
- `getProperties` - Right after retrieving the number of devices, NCCL queries the properties for each available network device. These properties are necessary when multiple adapters are present to ensure NCCL uses each adapter in the optimal way.
  - The `name` is only used for logging.
  - The `pciPath` is the base for all topology detection and should point to the PCI device directory in `/sys`. This is typically the directory pointed to by `/sys/class/net/eth0/device` or `/sys/class/infiniband/mlx5_0/device`. If the network interface is virtual, then `pciPath` should be `NULL`.
  - The `guid` field is used to determine whether network adapters are connected to multiple PCI endpoints. For normal cases, this is set to the device number. If multiple network devices have the same `guid`, then NCCL understands them to be sharing the same network port to the fabric. In this case, it will not use the port multiple times.
  - The `ptrSupport` field indicates whether or not CUDA pointers are supported. If so, it should be set to `NCCL_PTR_HOST|NCCL_PTR_CUDA`. Otherwise, it should be set to `NCCL_PTR_HOST`. If the plugin supports `dmabuf`, it should set `ptrSupport` to `NCCL_PTR_HOST|NCCL_PTR_CUDA|NCCL_PTR_DMABUF` and provide a `regMrDmaBuf` function.

  - The `regIsGlobal` field allows NCCL to register buffers in advance, for example, using a loopback connection. Later, it also lets NCCL expect that a subsequent registration on a buffer from a previous registration will happen nearly immediately, because the buffer is already known by the network adapter. A typical implementation maintains a registration cache, with the call to `ncclCommRegister` creating the initial entry in the cache using `regMr()` on a loopback connection. Any later call to the NCCL system can call `regMr()` again on the real connection, with the real buffer (which could be at a different offset within the original buffer, with a smaller size, for example). It could then call `deregMr()` immediately afterwards. The `ncclCommDeregister` call should issue the final call to `deregMr()` and effectively remove the mapping on the network adapter.
  - The `speed` field indicates the speed of the network port in Mbps (10^6 bits per second). This ensures proper optimization of flows within the node.
  - The `port` field indicates the port number. This is important for topology detection and flow optimization within the node when a NIC with a single PCI connection is connected to the fabric through multiple ports.
  - The `latency` field indicates the network latency in microseconds. This can be useful to improve the NCCL tuning and ensure NCCL switches from tree to ring at the correct size.
  - The `maxComms` field indicates the maximum number of connections that can be created.
  - The `maxRecvs` field indicates the maximum number for grouped receive operations (see grouped receive).

Connection establishment

Connections are used in an unidirectional manner, with a sender side and a receiver side.

- `listen` - To create a connection, NCCL calls `listen` on the receiver side. This function accepts a device number as an input argument and returns a local `listenComm` object and a `handle` to pass to the other side of the connection, so that the sender can connect to the receiver. The `handle` is a buffer of size `NCCL_NET_HANDLE_MAXSIZE` and is provided by NCCL. This call should never block, but unlike `connect` and `accept`, `listenComm` should never be `NULL` if the call succeeds.
- `connect` - NCCL uses its bootstrap infrastructure to provide the `handle` to the sender side, then calls `connect` on the sender side on a given device index `dev` and provides the `handle`. `connect` should not block either. Instead, it sets `sendComm` to `NULL` and returns `ncclSuccess`. In that case, NCCL will keep calling `accept` again until it succeeds.
- `accept` - To finalize the connection, the receiver side calls `accept` on the `listenComm` object previously returned by the `listen` call. If the sender did not connect yet, `accept` should not block. It should return `ncclSuccess`, setting `recvComm` to `NULL`. NCCL will keep calling `accept` again until it succeeds.
- `closeListen` / `closeSend` / `closeRecv` - When a `listenComm`, `sendComm`, or `recvComm` object is no longer needed, NCCL calls `closeListen`, `closeSend`, or `closeRecv` to free the associated resources.

Communication

Communication is handled using the asynchronous send and receive operations: `isend`, `irecv`, and `test`. To support RDMA capabilities, buffer registration and flush functions are provided.

To keep track of asynchronous send, receive, and flush operations, requests are returned to NCCL, then queried using `test`. Each `sendComm` or `recvComm` must be able to handle `NCCL_NET_MAX_REQUESTS` requests in parallel.

Note: This value should be multiplied by the multi-receive capability of the plugin for the sender side, so the plugin can effectively have `NCCL_NET_MAX_REQUESTS` multi-receive operations happening in parallel. If `maxRecvs` is 8

and NCCL_NET_MAX_REQUESTS is 8, then each sendComm must be able to handle up to 64 (8x8) concurrent isend operations.

- regMr - Prior to sending or receiving data, NCCL calls regMr with any buffers later used for communication. It provides a sendComm or recvComm object for the comm argument, the buffer pointer data, the size, and the type. The type is either NCCL_PTR_HOST or NCCL_PTR_CUDA if the network supports CUDA pointers.

  The network plugin can use the output argument mhandle to store any reference to the memory registration, because mhandle is returned for all isend, irecv, iflush, and deregMr calls.

- regMrDmaBuf - If the plugin has set the NCCL_PTR_DMABUF property in ptrSupport, NCCL uses regMrDmaBuf instead of regMr. If the property was not set, regMrDmaBuf can be set to NULL.

- deregMr - When buffers are no longer used for communication, NCCL calls deregMr to let the plugin free resources. This function is used to deregister handles returned by regMr and regMrDmaBuf.

- isend - Data is sent through the connection using isend, passing the sendComm object previously created by connect, the buffer described by data, size, and mhandle. A tag must be used if the network supports multi-receive operations (see irecv) to distinguish between different send requests matching the same multi-receive. Otherwise it can be set to 0.

  The isend operation returns a handle in the request argument for further calls to test. If the isend operation cannot be initiated, request is set to NULL. NCCL will call isend again later.

- irecv - To receive data, NCCL calls irecv with the recvComm returned by accept. The argument n configures NCCL for multi-receive, to allow grouping of multiple sends through a single network connection. Each buffer can be described by the data, sizes, and mhandles arrays. tags specify a tag for each receive so that each of the n independent isend operations is received into the right buffer.

  If all receive operations can be initiated, irecv returns a handle in the request pointer. Otherwise, it sets the pointer to NULL. In the case of multi-receive, all n receive operations are handled by a single request handle.

  The sizes provided to irecv can (and will) be larger than the size of the isend operation. However, it is an error if the receive size is smaller than the send size.

  Note: For a given connection, send and receive operations should always match in the order they were posted. Tags provided for receive operations are only used to assign a given send operation to one of the buffers of the first (multi-)receive operation in the queue, not to allow for out-of-order tag matching on any receive operation posted.

- test - After an isend or irecv operation is initiated, NCCL calls test on the request handles until the operation completes. When that happens, done is set to 1 and sizes is set to the real size sent or received, the latter could potentially be lower than the size passed to irecv.

  In the case of a multi-receive, all receives are considered as part of a single operation, the goal being to allow aggregation. Therefore, they share a single request and a single done status. However, they can have different sizes, so if done is non-zero, the sizes array should contain the n sizes corresponding to the buffers passed to irecv.

  After test returns 1 in done, the request handle can be freed. This means that NCCL will never call test again on that request, unless it is reallocated by another call to isend or irecv.

- iflush - After a receive operation completes, if the operation was targeting GPU memory and received a non-zero number of bytes, NCCL calls iflush. This lets the network flush any buffer to ensure the GPU can read it immediately without seeing stale data. This flush operation is decoupled from the test code to improve the latency of LL* protocols, because those are capable of determining when data is valid or not.

  iflush returns a request which must be queried using test until it completes.

CHAPTER

SEVEN

TROUBLESHOOTING RCCL

This topic explains the steps to troubleshoot functional and performance issues with RCCL. While debugging, collect the output from the commands in this guide. This data can be used as supporting information when submitting an issue report to AMD.

7.1 Collecting system information

Collect this information about the ROCm version, GPU/accelerator, platform, and configuration.

- Verify the ROCm version. This might be a release version or a mainline or staging version. Use this command to display the version:

```
cat /opt/rocm/.info/version
```

  Run the following command and collect the output:

```
rocm_agent_enumerator
```

  Also, collect the name of the GPU or accelerator:

```
rocminfo
```

- Run these `amd-smi` commands to display the system topology.

```
amd-smi
amd-smi topology
amd-smi static --driver
amd-smi firmware
amd-smi xgmi
```

- Determine the values of the `PATH` and `LD_LIBRARY_PATH` environment variables.

```
echo $PATH
echo $LD_LIBRARY_PATH
```

- Collect the HIP configuration.

```
/opt/rocm/bin/hipconfig --full
```

- Verify the network settings and setup. Use the `ibv_devinfo` command to display information about the available RDMA devices and determine whether they are installed and functioning properly. Run `rdma link` to print a summary of the network links.

```
ibv_devinfo
rdma link
```

7.1.1 Isolating the issue

The problem might be a general issue or specific to the architecture or system. To narrow down the issue, collect information about the GPU or accelerator and other details about the platform and system. Some issues to consider include:

- Is ROCm running on:
  - A bare-metal setup
  - In a Docker container (determine the name of the Docker image)
  - In an SR-IOV virtualized
  - Some combination of these configurations
- Is the problem only seen on a specific GPU architecture?
- Is it only seen on a specific system type?
- Is it happening on a single node or multinode setup?
- Use the following troubleshooting techniques to attempt to isolate the issue.
  - Build or run the develop branch version of RCCL and see if the problem persists.
  - Try an earlier RCCL version (minor or major).
  - If you recently changed the ROCm runtime configuration, AMD Kernel-mode GPU Driver (KMD), or compiler, rerun the test with the previous configuration.

7.2 Collecting RCCL information

Collect the following information about the RCCL installation and configuration.

- Run the `ldd` command to list any dynamic dependencies for RCCL.

```
ldd <specify-path-to-librccl.so>
```

- Determine the RCCL version. This might be the pre-packaged component in `/opt/rocm/lib` or a version that was built from source. To verify the RCCL version, enter the following command, then run either rccl-tests or an e2e application.

```
export NCCL_DEBUG=VERSION
```

- Run rccl-tests and collect the results. For information on how to build and run rccl-tests, see the rccl-tests GitHub.
- Collect the RCCL logging information. Enable the debug logs, then run rccl-tests or any e2e workload to collect the logs. Use the following command to enable the logs.

```
export NCCL_DEBUG=INFO
```

7.2.1 Using the RCCL Replayer

The RCCL Replayer is a debugging tool designed to analyze and replay the collective logs obtained from RCCL runs. It can be helpful when trying to reproduce problems, because it uses dummy data and doesn't have any dependencies on non-RCCL calls. For more information, see RCCL Replayer GitHub documentation.

You must build the RCCL Replayer before you can use it. To build it, run these commands. Ensure `MPI_DIR` is set to the path where MPI is installed.

```
cd rccl/tools/rccl_replayer
MPI_DIR=/path/to/mpi make
```

To use the RCCL Replayer, follow these steps:

1. Collect the per-rank logs from the RCCL run by adding the following environment variables. This prevents any race conditions that might cause ranks to interrupt the output from other ranks.

   ```
   NCCL_DEBUG=INFO NCCL_DEBUG_SUBSYS=COLL NCCL_DEBUG_FILE=some_name_here.%h.%p.log
   ```

2. Combine all the logs into a single file. This will become the input to the RCCL Replayer.

   ```
   cat some_name_here_*.log > some_name_here.log
   ```

3. Run the RCCL Replayer using the following command. Replace `<numProcesses>` with the number of MPI processes to run, `</path/to/logfile>` with the path to the collective log file generated during the RCCL runs, and `<numGpusPerMpiRank>` with the number of GPUs per MPI rank used in the application.

   ```
   mpirun -np <numProcesses> ./rcclReplayer </path/to/logfile> <numGpusPerMpiRank>
   ```

   In a multi-node application environment, you can replay the collective logs on multiple nodes using the following command:

   ```
   mpirun --hostfile <path/to/hostfile.txt> -np <numProcesses> ./rcclReplayer </path/
   ↪to/logfile> <numGpusPerMpiRank>
   ```

   Note: Depending on the MPI library you're using, you might need to modify the `mpirun` command.

7.3 Analyzing performance issues

If the issues involve performance issues in an e2e workload, try the following microbenchmarks and collect the results. Follow the instructions in the subsequent sections to run these benchmarks and provide the results to the support team.

- TransferBench
- RCCL Unit Tests
- rccl-tests

7.3.1 Collect the TransferBench data

TransferBench allows you to benchmark simultaneous copies between user-specified devices. For more information, see the TransferBench documentation.

To collect the TransferBench data, follow these steps:

1. Clone the TransferBench Git repository.

```
git clone https://github.com/ROCm/TransferBench.git
```

2. Change to the new directory and build the component.

```
cd TransferBench
make
```

3. Run the TransferBench utility with the following parameters and save the results.

```
USE_FINE_GRAIN=1 GFX_UNROLL=2 ./TransferBench a2a 64M 8
```

7.3.2 Collect the RCCL microbenchmark data

To use the RCCL tests to collect the RCCL benchmark data, follow these steps:

1. Disable NUMA auto-balancing using the following command:

```
sudo sysctl kernel.numa_balancing=0
```

   Run the following command to verify the setting. The expected output is `0`.

```
cat /proc/sys/kernel/numa_balancing
```

2. Build MPI, RCCL, and rccl-tests. To download and install MPI, see either OpenMPI or MPICH. To learn how to build and run rccl-tests, see the rccl-tests GitHub.
3. Run rccl-tests with MPI and collect the performance numbers.

7.4 RCCL and NCCL comparisons

If you are also using NVIDIA hardware or NCCL and notice a performance gap between the two systems, collect the system and performance data on the NVIDIA platform. Provide both sets of data to the support team.

CHAPTER

EIGHT

RCCL USAGE TIPS

This topic describes some of the more common RCCL extensions, such as NPKit and MSCCL, and provides tips on how to configure and customize the application.

8.1 NPKit

RCCL integrates NPKit, a profiler framework that enables the collection of fine-grained trace events in RCCL components, especially in giant collective GPU kernels. See the NPKit sample workflow for RCCL for a fully-automated usage example. It also provides useful templates for the following manual instructions.

To manually build RCCL with NPKit enabled, pass `-DNPKIT_FLAGS="-DENABLE_NPKIT -DENABLE_NPKIT_... (other NPKit compile-time switches)"` to the `cmake` command. All NPKit compile-time switches are declared in the RCCL code base as macros with the prefix `ENABLE_NPKIT_`. These switches control the information that is collected.

Note: NPKit only supports the collection of non-overlapped events on the GPU. The `-DNPKIT_FLAGS` settings must follow this rule.

To manually run RCCL with NPKit enabled, set the environment variable `NPKIT_DUMP_DIR` to the NPKit event dump directory. NPKit only supports one GPU per process. To manually analyze the NPKit dump results, use npkit_trace_generator.py.

8.2 MSCCL/MSCCL++

RCCL integrates MSCCL and MSCCL++ to leverage these highly efficient GPU-GPU communication primitives for collective operations. Microsoft Corporation collaborated with AMD for this project.

MSCCL uses XMLs for different collective algorithms on different architectures. RCCL collectives can leverage these algorithms after the user provides the corresponding XML. The XML files contain sequences of send-recv and reduction operations for the kernel to run.

MSCCL is enabled by default on the AMD Instinct™ MI300X accelerator. On other platforms, users might have to enable it using the setting `RCCL_MSCCL_FORCE_ENABLE=1`. By default, MSCCL is only used if every rank belongs to a unique process. To disable this restriction for multi-threaded or single-threaded configurations, use the setting `RCCL_MSCCL_ENABLE_SINGLE_PROCESS=1`.

RCCL allreduce and allgather collectives can leverage the efficient MSCCL++ communication kernels for certain message sizes. MSCCL++ support is available whenever MSCCL support is available. To run a RCCL workload with MSCCL++ support, set the following RCCL environment variable:

```
RCCL_MSCCLPP_ENABLE=1
```

To set the message size threshold for using MSCCL++, use the environment variable `RCCL_MSCCLPP_THRESHOLD`, which has a default value of 1MB. After `RCCL_MSCCLPP_THRESHOLD` has been set, RCCL invokes MSCCL++ kernels for all message sizes less than or equal to the specified threshold.

The following restrictions apply when using MSCCL++. If these restrictions are not met, operations fall back to using MSCCL or RCCL.

- The message size must be a non-zero multiple of 32 bytes
- It does not support `hipMallocManaged` buffers
- Allreduce only supports the `float16`, `int32`, `uint32`, `float32`, and `bfloat16` data types
- Allreduce only supports the sum operation

8.3 Enabling peer-to-peer transport

To enable peer-to-peer access on machines with PCIe-connected GPUs, set the HSA environment variable as follows:

```
HSA_FORCE_FINE_GRAIN_PCIE=1
```

This feature requires GPUs that support peer-to-peer access along with proper large BAR addressing support.

8.4 Ignoring CPU affinity with multi-node

Depending on the job launcher and the requirements of your workload, performance as the communication workload scales can be improved by setting `NCCL_IGNORE_CPU_AFFINITY`. This allows the RCCL communication library to ignore the job's supplied CPU affinity and use the GPU affinity only.

```
NCCL_IGNORE_CPU_AFFINITY=1
```

For general usage, this environment variable is not set so it doesn't interfere with the user or launcher supplied preferences.

8.5 Improving performance on the MI300X

This section outlines ways to improve RCCL performance on MI300X systems, including guidelines for systems with fewer than eight GPUs and the most efficient GPU partition modes.

8.5.1 Configuration with fewer than eight GPUs

On a system with eight MI300X accelerators, each pair of accelerators is connected with dedicated Infinity Fabric™ links in a fully connected topology. For collective operations, this can achieve good performance when all eight accelerators (and all Infinity Fabric links) are used. When fewer than eight GPUs are used, however, this can only achieve a fraction of the potential bandwidth on the system. However, if your workload warrants using fewer than eight MI300X accelerators on a system, you can set the run-time variable `NCCL_MIN_NCHANNELS` to increase the number of channels. For example:

```
export NCCL_MIN_NCHANNELS=32
```

Increasing the number of channels can benefit performance, but it also increases GPU utilization for collective operations. Additionally, RCCL pre-defines a higher number of channels when only two or four accelerators are in use on a 8*MI300X system. In this situation, RCCL uses 32 channels with two MI300X accelerators and 24 channels for four MI300X accelerators.

8.5.2 NPS4 and CPX partition modes

The term compute partitioning modes, or Modular Chiplet Platform (MCP), refers to the logical partitioning of XCDs into devices in the ROCm stack. The names are derived from the number of logical partitions that are created out of the eight XCDs. In the default mode, SPX (Single Partition X-celerator), all eight XCDs are viewed as a single logical compute element, meaning that the amd-smi utility will show a single MI300X device. In CPX (Core Partitioned X-celerator) mode, each XCD appears as a separate logical GPU, for example, as eight separate GPUs in amd-smi per MI300X. CPX mode can be viewed as having explicit scheduling privileges for each individual compute element (XCD).

While compute partitioning modes change the space on which you can assign work to compute units, the memory partitioning modes (known as Non-Uniform Memory Access (NUMA) Per Socket (NPS)) change the number of NUMA domains that a device exposes. In other words, it changes the number of HBM stacks which are accessible to a compute unit, and therefore the size of its memory space. However, for the MI300X, the number of memory partitions must be less than or equal to the number of compute partitions. NPS4 (viewing pairs of HBM stacks as a disparate element), for example, is only enabled when in CPX mode (viewing each XCD as a disparate element).

- Compute partition modes
  - In SPX mode, workgroups launched to the device are distributed round-robin to the XCDs in the device, meaning that the programmer cannot have explicit control over which XCD a workgroup is assigned to.
  - In CPX mode, workgroups are launched to a single XCD, meaning the programmer has explicit control over work placement onto the XCDs.
- Memory partition modes
  - In NPS1 mode (compatible with CPX and SPX), the entire memory is accessible to all XCDs.
  - In NPS4 mode (compatible with CPX), each memory quadrant of the memory is directly visible to the logical devices in its quadrant. An XCD can still access all portions of memory through multi-GPU programming techniques.

The MI300 CPX mode can be accessed using the following AMD SMI documentation commands.

```
amd-smi set --gpu all --compute-partition CPX
amd-smi set --gpu all --memory-partition NPS4
```

RCCL performance with CPX and NPS4

To run RCCL allreduce on 64 GPUs with CPX+NPS4 mode on the MI300X, use this example:

```
mpirun -np 64 --bind-to numa rccl-tests/build/all_reduce_perf -b 8 -e 1G -f 2 -g 1
```

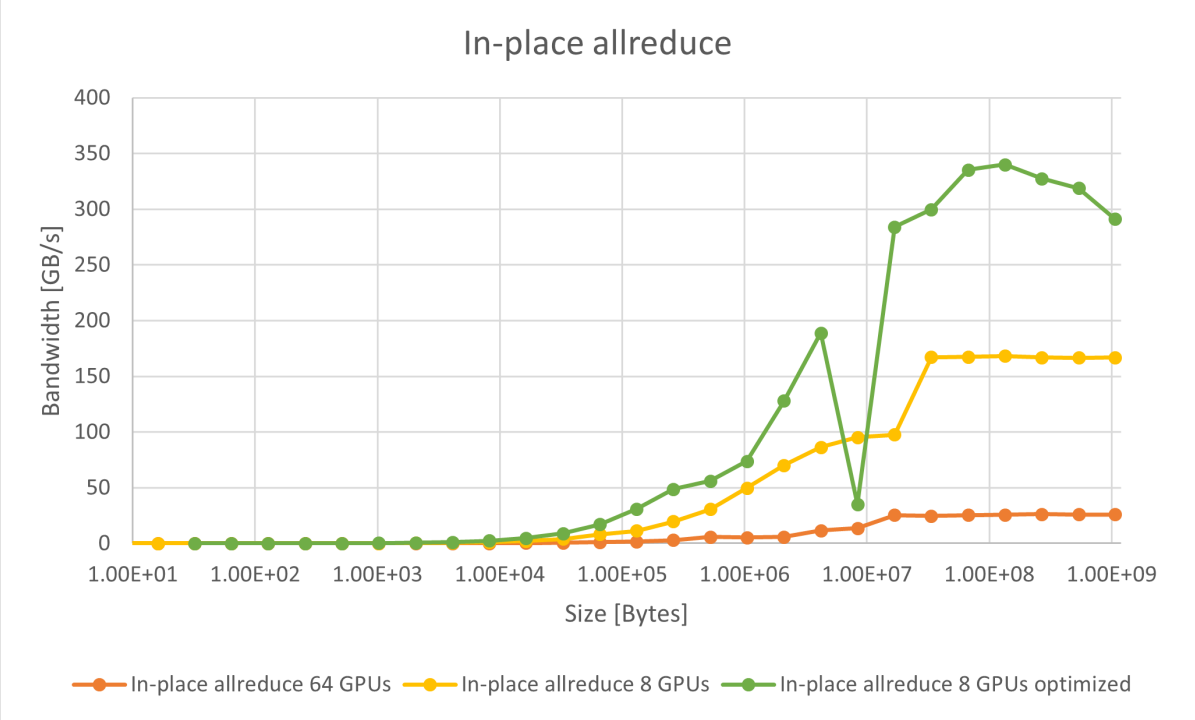
To run RCCL allreduce on 8 GPUs in the same OAM with CPX+NPS4 mode on the MI300X, use this example:

```
export ROCR_VISIBLE_DEVICES=0,1,2,3,4,5,6,7

mpirun -np 8 --bind-to numa rccl-tests/build/all_reduce_perf -b 8 -e 1G -f 2 -g 1
```

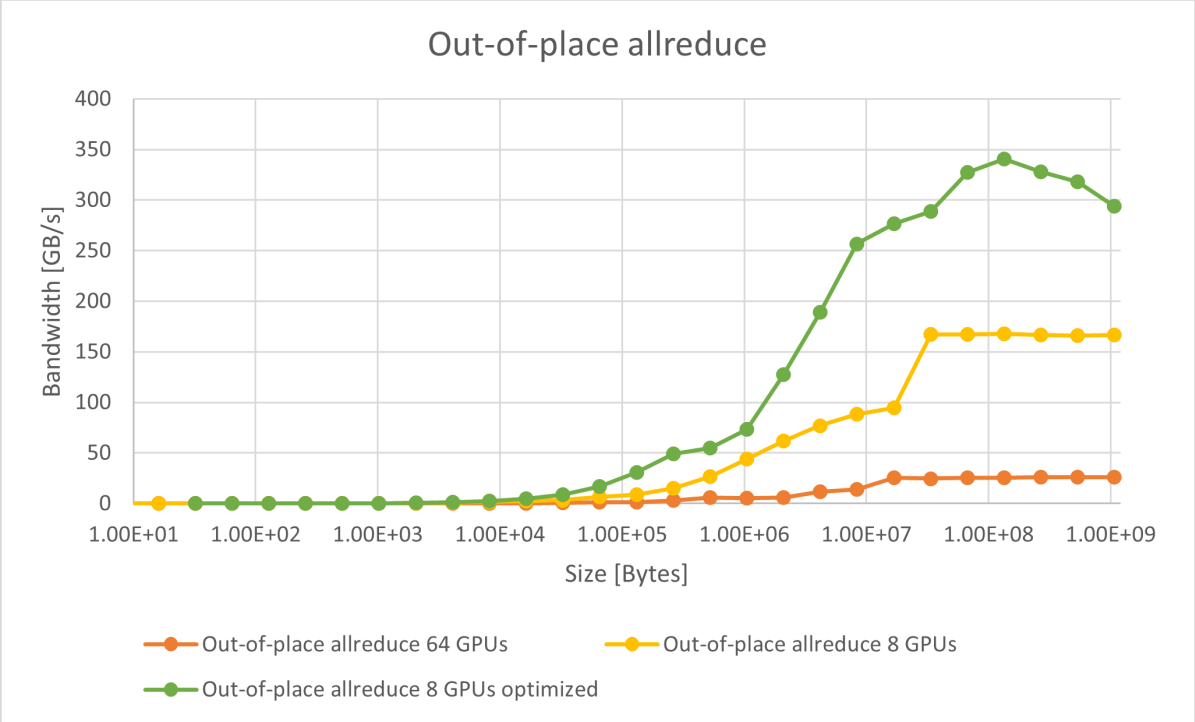
RCCL delivers improved allreduce performance in CPX mode for TP=8 (8 GPUs in the same OAM) on the MI300X.

```
export HIP_FORCE_DEV_KERNARG=1
export RCCL_MSCCLPP_THRESHOLD=1073741824

export MSCCLPP_READ_ALLRED=1
export ROCR_VISIBLE_DEVICES=0,1,2,3,4,5,6,7

mpirun -np 8 --bind-to numa rccl-tests/build/all_reduce_perf -b 32 -e 1G -f 2 -g 1 -G 2 -
↪w 20 -n 50
```

Here are the benchmark results for in-place (where the output buffer is used as the input buffer) and out-of-place allreduce bus bandwidth.



A significant performance improvement is achievable with optimized CPX mode, which peaks at ~340 GB/s with a single OAM. The difference in bus bandwidth between the unoptimized and optimized modes increases as the buffer size grows.



Using RCCL and CPX in PyTorch

The PyTorch all_reduce benchmark is used to reproduce the performance reported by RCCL-Tests with the RCCL and CPX optimizations.

Note: To use RCCL with CPX mode in PyTorch, check the RCCL version used by PyTorch.

For a virtualenv with a .whl-based PyTorch setup (such as nightly/rocm6.2), this would be in `<path-to-your-venv>/lib/<python-version>/site-packages/torch/lib/librccl.so` This is the version of RCCL that is packaged as part of ROCm version 6.2.

RCCL for CPX mode was enabled in ROCm 6.3.0. To use the CPX features, replace the existing `librccl.so` with one from ROCm 6.3.0 or newer or from a local build of the RCCL develop branch.

To test the effects of RCCL on PyTorch, the stas00 all reduce benchmark was used. The following command is used to run a single OAM allreduce benchmark:

```
export ROCR_VISIBLE_DEVICES=0,1,2,3,4,5,6,7
python -u -m torch.distributed.run --nproc_per_node=8 --rdzv_endpoint localhost:6000  --
↪rdzv_backend c10d all_reduce_bench.py
```

For better performance, the `HIP_FORCE_DEV_KERNARG`, `RCCL_MSCCLPP_THRESHOLD`, and `TORCH_NCCL_USE_TENSOR_REGISTER_ALLOCATOR_HOOK` environment variables are set during the benchmark in the following manner:

```
export TORCH_NCCL_USE_TENSOR_REGISTER_ALLOCATOR_HOOK=1
export HIP_FORCE_DEV_KERNARG=1
export RCCL_MSCCLPP_THRESHOLD=$((2*1024*1024*1024))
export MSCCLPP_READ_ALLRED=1
```

(continues on next page)

(continued from previous page)

```
export ROCR_VISIBLE_DEVICES=0,1,2,3,4,5,6,7
python -u -m torch.distributed.run --nproc_per_node=8 --rdzv_endpoint localhost:6000  --
↪rdzv_backend c10d all_reduce_bench.py
```

The default allreduce PyTorch benchmark peak bus bandwidth performance is ~170 GB/s on a single OAM with ROCm 6.2.4, while the optimized run for CPX on a single OAM peaks at ~315 GB/s.

8.5.3 Context tracking on GPUs

Context tracking is disabled by default for optimal performance. However, enabling of context tracking can significantly improve performance in certain scenarios. To enable context tracking, set the following environment variable:

```
export RCCL_ENABLE_CONTEXT_TRACKING=1
```

CHAPTER

NINE

RCCL LIBRARY SPECIFICATION

This document provides details of the API library.

9.1 Communicator functions

ncclResult_t **ncclGetUniqueId**(*ncclUniqueId* *uniqueId)

Generates an ID for ncclCommInitRank.

Generates an ID to be used in ncclCommInitRank. ncclGetUniqueId should be called once by a single rank and the ID should be distributed to all ranks in the communicator before using it as a parameter for ncclCommInitRank.

Parameters

uniqueId – **[out]** Pointer to where uniqueId will be stored

Returns

Result code. See *Result Codes* for more details.

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

Creates a new communicator (multi thread/process version).

Rank must be between 0 and nranks-1 and unique within a communicator clique. Each rank is associated to a CUDA device, which has to be set before calling ncclCommInitRank. ncclCommInitRank implicitly syncronizes with other ranks, so it must be called by different threads/processes or use ncclGroupStart/ncclGroupEnd.

Parameters

- **comm** – **[out]** Pointer to created communicator
- **nranks** – **[in]** Total number of ranks participating in this communicator
- **commId** – **[in]** UniqueId required for initialization
- **rank** – **[in]** Current rank to create communicator for

Returns

Result code. See *Result Codes* for more details.

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

Creates a clique of communicators (single process version).

This is a convenience function to create a single-process communicator clique. Returns an array of ndev newly initialized communicators in comm. comm should be pre-allocated with size at least ndev*sizeof(ncclComm_t). If devlist is NULL, the first ndev HIP devices are used. Order of devlist defines user-order of processors within the communicator.

Parameters

- **comm** – **[out]** Pointer to array of created communicators
- **ndev** – **[in]** Total number of ranks participating in this communicator
- **devlist** – **[in]** Array of GPU device indices to create for

Returns
Result code. See *Result Codes* for more details.

ncclResult_t **ncclCommDestroy**(*ncclComm_t* comm)

Frees local resources associated with communicator object.

Destroy all local resources associated with the passed in communicator object

Parameters
comm – **[in]** Communicator to destroy

Returns
Result code. See *Result Codes* for more details.

ncclResult_t **ncclCommAbort**(*ncclComm_t* comm)

Abort any in-progress calls and destroy the communicator object.

Frees resources associated with communicator object and aborts any operations that might still be running on the device.

Parameters
comm – **[in]** Communicator to abort and destroy

Returns
Result code. See *Result Codes* for more details.

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

Gets the number of ranks in the communicator clique.

Returns the number of ranks in the communicator clique (as set during initialization)

Parameters

- **comm** – **[in]** Communicator to query
- **count** – **[out]** Pointer to where number of ranks will be stored

Returns
Result code. See *Result Codes* for more details.

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

Get the ROCm device index associated with a communicator.

Returns the ROCm device number associated with the provided communicator.

Parameters

- **comm** – **[in]** Communicator to query
- **device** – **[out]** Pointer to where the associated ROCm device index will be stored

Returns
Result code. See *Result Codes* for more details.

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

Get the rank associated with a communicator.

Returns the user-ordered "rank" associated with the provided communicator.

Parameters

- **comm** – **[in]** Communicator to query
- **rank** – **[out]** Pointer to where the associated rank will be stored

Returns
Result code. See *Result Codes* for more details.

9.2 Collective communication operations

Collective communication operations must be called separately for each communicator in a communicator clique.

They return when operations have been enqueued on the hipstream.

Since they may perform inter-CPU synchronization, each call has to be done from a different thread or process, or need to use Group Semantics (see below).

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

Reduce.

Reduces data arrays of length *count* in *sendbuff* into *recvbuff* using *op* operation. recvbuff* may be NULL on all calls except for root device. root* is the rank (not the HIP device) where data will reside after the operation is complete. In-place operation will happen if sendbuff == recvbuff.

Parameters

- **sendbuff** – **[in]** Local device data buffer to be reduced
- **recvbuff** – **[out]** Data buffer where result is stored (only for *root* rank). May be null for other ranks.
- **count** – **[in]** Number of elements in every send buffer
- **datatype** – **[in]** Data buffer element datatype
- **op** – **[in]** Reduction operator type
- **root** – **[in]** Rank where result data array will be stored
- **comm** – **[in]** Communicator group object to execute on
- **stream** – **[in]** HIP stream to execute collective on

Returns
Result code. See *Result Codes* for more details.

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

(Deprecated) Broadcast (in-place)

Copies *count* values from *root* to all other devices. root is the rank (not the CUDA device) where data resides before the operation is started. This operation is implicitly in-place.

Parameters

- **buff** – **[inout]** Input array on *root* to be copied to other ranks. Output array for all ranks.
- **count** – **[in]** Number of elements in data buffer
- **datatype** – **[in]** Data buffer element datatype
- **root** – **[in]** Rank owning buffer to be copied to others
- **comm** – **[in]** Communicator group object to execute on

- **stream** – **[in]** HIP stream to execute collective on

Returns
Result code. See *Result Codes* for more details.

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

Broadcast.

Copies *count* values from *sendbuff* on *root* to *recvbuff* on all devices. root* is the rank (not the HIP device) where data resides before the operation is started. sendbuff* may be NULL on ranks other than *root*. In-place operation will happen if *sendbuff* == *recvbuff*.

Parameters

- **sendbuff** – **[in]** Data array to copy (if *root*). May be NULL for other ranks
- **recvbuff** – **[in]** Data array to store received array
- **count** – **[in]** Number of elements in data buffer
- **datatype** – **[in]** Data buffer element datatype
- **root** – **[in]** Rank of broadcast root
- **comm** – **[in]** Communicator group object to execute on
- **stream** – **[in]** HIP stream to execute collective on

Returns
Result code. See *Result Codes* for more details.

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

All-Reduce.

Reduces data arrays of length *count* in *sendbuff* using *op* operation, and leaves identical copies of result on each *recvbuff*. In-place operation will happen if sendbuff == recvbuff.

Parameters

- **sendbuff** – **[in]** Input data array to reduce
- **recvbuff** – **[out]** Data array to store reduced result array
- **count** – **[in]** Number of elements in data buffer
- **datatype** – **[in]** Data buffer element datatype
- **op** – **[in]** Reduction operator
- **comm** – **[in]** Communicator group object to execute on
- **stream** – **[in]** HIP stream to execute collective on

Returns
Result code. See *Result Codes* for more details.

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

Reduce-Scatter.

Reduces data in *sendbuff* using *op* operation and leaves reduced result scattered over the devices so that *recvbuff* on rank i will contain the i-th block of the result. Assumes sendcount is equal to nranks*recvcount, which means that *sendbuff* should have a size of at least nranks*recvcount elements. In-place operations will happen if recvbuff == sendbuff + rank * recvcount.

Parameters

- **sendbuff** – **[in]** Input data array to reduce
- **recvbuff** – **[out]** Data array to store reduced result subarray
- **recvcount** – **[in]** Number of elements each rank receives
- **datatype** – **[in]** Data buffer element datatype
- **op** – **[in]** Reduction operator
- **comm** – **[in]** Communicator group object to execute on
- **stream** – **[in]** HIP stream to execute collective on

Returns

Result code. See *Result Codes* for more details.

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

All-Gather.

Each device gathers *sendcount* values from other GPUs into *recvbuff*, receiving data from rank i at offset i*sendcount. Assumes recvcount is equal to nranks*sendcount, which means that recvbuff should have a size of at least nranks*sendcount elements. In-place operations will happen if sendbuff == recvbuff + rank * sendcount.

Parameters

- **sendbuff** – **[in]** Input data array to send
- **recvbuff** – **[out]** Data array to store the gathered result
- **sendcount** – **[in]** Number of elements each rank sends
- **datatype** – **[in]** Data buffer element datatype
- **comm** – **[in]** Communicator group object to execute on
- **stream** – **[in]** HIP stream to execute collective on

Returns

Result code. See *Result Codes* for more details.

ncclResult_t **ncclSend**(const void *sendbuff, size_t count, *ncclDataType_t* datatype, int peer, *ncclComm_t* comm, hipStream_t stream)

Send.

Send data from *sendbuff* to rank *peer*. Rank *peer* needs to call ncclRecv with the same *datatype* and the same *count* as this rank. This operation is blocking for the GPU. If multiple ncclSend and ncclRecv operations need to progress concurrently to complete, they must be fused within a ncclGroupStart / ncclGroupEnd section.

Parameters

- **sendbuff** – **[in]** Data array to send
- **count** – **[in]** Number of elements to send
- **datatype** – **[in]** Data buffer element datatype
- **peer** – **[in]** Peer rank to send to
- **comm** – **[in]** Communicator group object to execute on
- **stream** – **[in]** HIP stream to execute collective on

Returns
Result code. See *Result Codes* for more details.

ncclResult_t **ncclRecv**(void *recvbuff, size_t count, *ncclDataType_t* datatype, int peer, *ncclComm_t* comm, hipStream_t stream)

Receive.

Receive data from rank *peer* into *recvbuff*. Rank *peer* needs to call ncclSend with the same datatype and the same count as this rank. This operation is blocking for the GPU. If multiple ncclSend and ncclRecv operations need to progress concurrently to complete, they must be fused within a ncclGroupStart/ ncclGroupEnd section.

Parameters

- **recvbuff** – **[out]** Data array to receive
- **count** – **[in]** Number of elements to receive
- **datatype** – **[in]** Data buffer element datatype
- **peer** – **[in]** Peer rank to send to
- **comm** – **[in]** Communicator group object to execute on
- **stream** – **[in]** HIP stream to execute collective on

Returns
Result code. See *Result Codes* for more details.

ncclResult_t **ncclGather**(const void *sendbuff, void *recvbuff, size_t sendcount, *ncclDataType_t* datatype, int root, *ncclComm_t* comm, hipStream_t stream)

Gather.

Root device gathers *sendcount* values from other GPUs into *recvbuff*, receiving data from rank i at offset i*sendcount. Assumes recvcount is equal to nranks*sendcount, which means that *recvbuff* should have a size of at least nranks*sendcount elements. In-place operations will happen if sendbuff == recvbuff + rank * sendcount. recvbuff* may be NULL on ranks other than *root*.

Parameters

- **sendbuff** – **[in]** Data array to send
- **recvbuff** – **[out]** Data array to receive into on *root*.
- **sendcount** – **[in]** Number of elements to send per rank
- **datatype** – **[in]** Data buffer element datatype
- **root** – **[in]** Rank that receives data from all other ranks
- **comm** – **[in]** Communicator group object to execute on
- **stream** – **[in]** HIP stream to execute collective on

Returns
Result code. See *Result Codes* for more details.

ncclResult_t **ncclScatter**(const void *sendbuff, void *recvbuff, size_t recvcount, *ncclDataType_t* datatype, int root, *ncclComm_t* comm, hipStream_t stream)

Scatter.

Scattered over the devices so that recvbuff on rank i will contain the i-th block of the data on root. Assumes sendcount is equal to nranks*recvcount, which means that *sendbuff* should have a size of at least nranks*recvcount elements. In-place operations will happen if recvbuff == sendbuff + rank * recvcount.

Parameters

- **sendbuff** – **[in]** Data array to send (on *root* rank). May be NULL on other ranks.
- **recvbuff** – **[out]** Data array to receive partial subarray into
- **recvcount** – **[in]** Number of elements to receive per rank
- **datatype** – **[in]** Data buffer element datatype
- **root** – **[in]** Rank that scatters data to all other ranks
- **comm** – **[in]** Communicator group object to execute on
- **stream** – **[in]** HIP stream to execute collective on

Returns
Result code. See *Result Codes* for more details.

ncclResult_t **ncclAllToAll**(const void *sendbuff, void *recvbuff, size_t count, *ncclDataType_t* datatype, *ncclComm_t* comm, hipStream_t stream)

All-To-All.

Device (i) send (j)th block of data to device (j) and be placed as (i)th block. Each block for sending/receiving has *count* elements, which means that *recvbuff* and *sendbuff* should have a size of nranks*count elements. In-place operation is NOT supported. It is the user's responsibility to ensure that sendbuff and recvbuff are distinct.

Parameters

- **sendbuff** – **[in]** Data array to send (contains blocks for each other rank)
- **recvbuff** – **[out]** Data array to receive (contains blocks from each other rank)
- **count** – **[in]** Number of elements to send between each pair of ranks
- **datatype** – **[in]** Data buffer element datatype
- **comm** – **[in]** Communicator group object to execute on
- **stream** – **[in]** HIP stream to execute collective on

Returns
Result code. See *Result Codes* for more details.

9.3 Group semantics

When managing multiple GPUs from a single thread, and since NCCL collective calls may perform inter-CPU synchronization, we need to "group" calls for different ranks/devices into a single call.

Grouping NCCL calls as being part of the same collective operation is done using ncclGroupStart and ncclGroupEnd. ncclGroupStart will enqueue all collective calls until the ncclGroupEnd call, which will wait for all calls to be complete. Note that for collective communication, ncclGroupEnd only guarantees that the operations are enqueued on the streams, not that the operation is effectively done.

Both collective communication and ncclCommInitRank can be used in conjunction of ncclGroupStart/ncclGroupEnd.

ncclResult_t **ncclGroupStart**()

Group Start.

Start a group call. All calls to RCCL until ncclGroupEnd will be fused into a single RCCL operation. Nothing will be started on the HIP stream until ncclGroupEnd.

Returns
Result code. See *Result Codes* for more details.

ncclResult_t **ncclGroupEnd**()

Group End.

End a group call. Start a fused RCCL operation consisting of all calls since ncclGroupStart. Operations on the HIP stream depending on the RCCL operations need to be called after ncclGroupEnd.

Returns
Result code. See *Result Codes* for more details.

9.4 Library functions

ncclResult_t **ncclGetVersion**(int *version)

Return the RCCL_VERSION_CODE of RCCL in the supplied integer.

This integer is coded with the MAJOR, MINOR and PATCH level of RCCL.

Parameters
version – **[out]** Pointer to where version will be stored

Returns
Result code. See *Result Codes* for more details.

const char ***ncclGetErrorString**(*ncclResult_t* result)

Returns a string for each result code.

Returns a human-readable string describing the given result code.

Parameters
result – **[in]** Result code to get description for

Returns
String containing description of result code.

9.5 Types

There are few data structures that are internal to the library. The pointer types to these structures are given below. The user would need to use these types to create handles and pass them between different library functions.

typedef struct ncclComm ***ncclComm_t**

Opaque handle to communicator.

A communicator contains information required to facilitate collective communications calls

struct **ncclUniqueId**

Opaque unique id used to initialize communicators.

The *ncclUniqueId* must be passed to all participating ranks

9.6 Enumerations

This section provides all the enumerations used.

enum **ncclResult_t**

Result type.

Return codes aside from ncclSuccess indicate that a call has failed

Values:

enumerator **ncclSuccess**

No error

enumerator **ncclUnhandledCudaError**

Unhandled HIP error

enumerator **ncclSystemError**

Unhandled system error

enumerator **ncclInternalError**

Internal Error - Please report to RCCL developers

enumerator **ncclInvalidArgument**

Invalid argument

enumerator **ncclInvalidUsage**

Invalid usage

enumerator **ncclRemoteError**

Remote process exited or there was a network error

enumerator **ncclInProgress**

RCCL operation in progress

enumerator **ncclNumResults**

Number of result types

enum **ncclRedOp_t**

Reduction operation selector.

Enumeration used to specify the various reduction operations ncclNumOps is the number of built-in ncclRedOp_t values and serves as the least possible value for dynamic ncclRedOp_t values constructed by ncclRedOpCreate functions.

ncclMaxRedOp is the largest valid value for ncclRedOp_t and is defined to be the largest signed value (since compilers are permitted to use signed enums) that won't grow sizeof(ncclRedOp_t) when compared to previous RCCL versions to maintain ABI compatibility.

Values:

enumerator **ncclSum**

Sum

enumerator **ncclProd**

Product

enumerator **ncclMax**

Max

enumerator **ncclMin**

Min

enumerator **ncclAvg**

Average

enumerator **ncclNumOps**

Number of built-in reduction ops

enumerator **ncclMaxRedOp**

Largest value for ncclRedOp_t

enum **ncclDataType_t**

Data types.

Enumeration of the various supported datatype

Values:

enumerator **ncclInt8**

enumerator **ncclChar**

enumerator **ncclUint8**

enumerator **ncclInt32**

enumerator **ncclInt**

enumerator **ncclUint32**

enumerator **ncclInt64**

enumerator **ncclUint64**

enumerator **ncclFloat16**

enumerator **ncclHalf**

enumerator **ncclFloat32**

enumerator **ncclFloat**

enumerator **ncclFloat64**

enumerator **ncclDouble**

enumerator **ncclBfloat16**

enumerator **ncclFloat8e4m3**

enumerator **ncclFloat8e5m2**

enumerator **ncclNumTypes**

CHAPTER

TEN

API LIBRARY

struct **ncclConfig_t**

Communicator configuration.

Users can assign value to attributes to specify the behavior of a communicator

Public Members

size_t **size**

Should not be touched

unsigned int **magic**

Should not be touched

unsigned int **version**

Should not be touched

int **blocking**

Whether or not calls should block or not

int **cgaClusterSize**

Cooperative group array cluster size

int **minCTAs**

Minimum number of cooperative thread arrays (blocks)

int **maxCTAs**

Maximum number of cooperative thread arrays (blocks)

const char ***netName**

Force NCCL to use a specfic network

int **splitShare**

Allow communicators to share resources

int **trafficClass**

Traffic class

const char ***commName**

Name of the communicator

int **collnetEnable**

Check for collnet enablement

int **CTAPolicy**

CTA Policy

int **shrinkShare**

Shrink size

int **nvlsCTAs**

Number of NVLS cooperative thread arrays (blocks)

struct **ncclSimInfo_t**

Public Members

size_t **size**

unsigned int **magic**

unsigned int **version**

float **estimatedTime**

struct **ncclUniqueId**

Opaque unique id used to initialize communicators.

The *ncclUniqueId* must be passed to all participating ranks

Public Members

char **internal**[NCCL_UNIQUE_ID_BYTES]

Opaque array>

file **mainpage.txt**

file **nccl.h.in**

#include <hip/hip_runtime.h>#include <hip/hip_fp16.h>#include <limits.h>

Defines

NCCL_H_

NCCL_MAJOR

NCCL_MINOR

NCCL_PATCH

NCCL_SUFFIX

NCCL_VERSION_CODE

NCCL_VERSION(X, Y, Z)

RCCL_BFLOAT16

RCCL_FLOAT8

RCCL_GATHER_SCATTER

RCCL_ALLTOALLV

RCCL_ALLREDUCE_WITH_BIAS

NCCL_COMM_NULL

NCCL_UNIQUE_ID_BYTES

NCCL_CONFIG_UNDEF_INT

NCCL_CONFIG_UNDEF_PTR

NCCL_SPLIT_NOCOLOR

NCCL_UNDEF_FLOAT

NCCL_WIN_DEFAULT

NCCL_WIN_COLL_SYMMETRIC

NCCL_CTA_POLICY_DEFAULT

NCCL_CTA_POLICY_EFFICIENCY

NCCL_SHRINK_DEFAULT

NCCL_SHRINK_ABORT

NCCL_CONFIG_INITIALIZER

NCCL_SIM_INFO_INITIALIZER

Typedefs

typedef struct ncclComm *ncclComm_t

Opaque handle to communicator.

A communicator contains information required to facilitate collective communications calls

typedef struct ncclWindow *ncclWindow_t

typedef int mscclAlgoHandle_t

Opaque handle to MSCCL algorithm.

Enums

enum ncclResult_t

Result type.

Return codes aside from ncclSuccess indicate that a call has failed

Values:

enumerator ncclSuccess

No error

enumerator ncclUnhandledCudaError

Unhandled HIP error

enumerator ncclSystemError

Unhandled system error

enumerator ncclInternalError

Internal Error - Please report to RCCL developers

enumerator ncclInvalidArgument

Invalid argument

enumerator **ncclInvalidUsage**

Invalid usage

enumerator **ncclRemoteError**

Remote process exited or there was a network error

enumerator **ncclInProgress**

RCCL operation in progress

enumerator **ncclNumResults**

Number of result types

enum **ncclRedOp_dummy_t**

Dummy reduction enumeration.

Dummy reduction enumeration used to determine value for ncclMaxRedOp

Values:

enumerator **ncclNumOps_dummy**

enum **ncclRedOp_t**

Reduction operation selector.

Enumeration used to specify the various reduction operations ncclNumOps is the number of built-in ncclRedOp_t values and serves as the least possible value for dynamic ncclRedOp_t values constructed by ncclRedOpCreate functions.

ncclMaxRedOp is the largest valid value for ncclRedOp_t and is defined to be the largest signed value (since compilers are permitted to use signed enums) that won't grow sizeof(ncclRedOp_t) when compared to previous RCCL versions to maintain ABI compatibility.

Values:

enumerator **ncclSum**

Sum

enumerator **ncclProd**

Product

enumerator **ncclMax**

Max

enumerator **ncclMin**

Min

enumerator **ncclAvg**

Average

enumerator **ncclNumOps**

Number of built-in reduction ops

enumerator **ncclMaxRedOp**

Largest value for ncclRedOp_t

enum **ncclDataType_t**

Data types.

Enumeration of the various supported datatype

Values:

enumerator **ncclInt8**

enumerator **ncclChar**

enumerator **ncclUint8**

enumerator **ncclInt32**

enumerator **ncclInt**

enumerator **ncclUint32**

enumerator **ncclInt64**

enumerator **ncclUint64**

enumerator **ncclFloat16**

enumerator **ncclHalf**

enumerator **ncclFloat32**

enumerator **ncclFloat**

enumerator **ncclFloat64**

enumerator **ncclDouble**

enumerator **ncclBfloat16**

enumerator **ncclFloat8e4m3**

enumerator **ncclFloat8e5m2**

enumerator **ncclNumTypes**

enum **ncclScalarResidence_t**

Location and dereferencing logic for scalar arguments.

Enumeration specifying memory location of the scalar argument. Based on where the value is stored, the argument will be dereferenced either while the collective is running (if in device memory), or before the ncclRedOpCreate() function returns (if in host memory).

Values:

enumerator **ncclScalarDevice**

Scalar is in device-visible memory

enumerator **ncclScalarHostImmediate**

Scalar is in host-visible memory

Functions

ncclResult_t **ncclMemAlloc**(void **ptr, size_t size)

ncclResult_t **pncclMemAlloc**(void **ptr, size_t size)

ncclResult_t **ncclMemFree**(void *ptr)

ncclResult_t **pncclMemFree**(void *ptr)

ncclResult_t **ncclGetVersion**(int *version)

Return the RCCL_VERSION_CODE of RCCL in the supplied integer.

This integer is coded with the MAJOR, MINOR and PATCH level of RCCL.

Parameters
version – **[out]** Pointer to where version will be stored

Returns
Result code. See *Result Codes* for more details.

ncclResult_t **ncclGetUniqueId**(*ncclUniqueId* *uniqueId)

Generates an ID for ncclCommInitRank.

Generates an ID to be used in ncclCommInitRank. ncclGetUniqueId should be called once by a single rank and the ID should be distributed to all ranks in the communicator before using it as a parameter for ncclCommInitRank.

Parameters
uniqueId – **[out]** Pointer to where uniqueId will be stored

Returns
Result code. See *Result Codes* for more details.

ncclResult_t **ncclCommInitRankConfig**(*ncclComm_t* *comm, int nranks, *ncclUniqueId* commId, int rank, *ncclConfig_t* *config)

Create a new communicator with config.

Create a new communicator (multi thread/process version) with a configuration set by users. See *Communicator Configuration* for more details. Each rank is associated to a CUDA device, which has to be set before calling ncclCommInitRank.

Parameters

- **comm** – **[out]** Pointer to created communicator
- **nranks** – **[in]** Total number of ranks participating in this communicator
- **commId** – **[in]** UniqueId required for initialization
- **rank** – **[in]** Current rank to create communicator for. [0 to nranks-1]
- **config** – **[in]** Pointer to communicator configuration

Returns

Result code. See *Result Codes* for more details.

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

Creates a new communicator (multi thread/process version).

Rank must be between 0 and nranks-1 and unique within a communicator clique. Each rank is associated to a CUDA device, which has to be set before calling ncclCommInitRank. ncclCommInitRank implicitly syncronizes with other ranks, so it must be called by different threads/processes or use ncclGroupStart/ncclGroupEnd.

Parameters

- **comm** – **[out]** Pointer to created communicator
- **nranks** – **[in]** Total number of ranks participating in this communicator
- **commId** – **[in]** UniqueId required for initialization
- **rank** – **[in]** Current rank to create communicator for

Returns

Result code. See *Result Codes* for more details.

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

Creates a clique of communicators (single process version).

This is a convenience function to create a single-process communicator clique. Returns an array of ndev newly initialized communicators in comm. comm should be pre-allocated with size at least ndev*sizeof(ncclComm_t). If devlist is NULL, the first ndev HIP devices are used. Order of devlist defines user-order of processors within the communicator.

Parameters

- **comm** – **[out]** Pointer to array of created communicators
- **ndev** – **[in]** Total number of ranks participating in this communicator
- **devlist** – **[in]** Array of GPU device indices to create for

Returns

Result code. See *Result Codes* for more details.

ncclResult_t **ncclCommFinalize**(*ncclComm_t* comm)

Finalize a communicator.

ncclCommFinalize flushes all issued communications and marks communicator state as ncclInProgress. The state will change to ncclSuccess when the communicator is globally quiescent and related resources are freed; then, calling ncclCommDestroy can locally free the rest of the resources (e.g. communicator itself) without blocking.

Parameters
comm – **[in]** Communicator to finalize

Returns
Result code. See *Result Codes* for more details.

ncclResult_t **ncclCommDestroy**(*ncclComm_t* comm)

Frees local resources associated with communicator object.

Destroy all local resources associated with the passed in communicator object

Parameters
comm – **[in]** Communicator to destroy

Returns
Result code. See *Result Codes* for more details.

ncclResult_t **ncclCommAbort**(*ncclComm_t* comm)

Abort any in-progress calls and destroy the communicator object.

Frees resources associated with communicator object and aborts any operations that might still be running on the device.

Parameters
comm – **[in]** Communicator to abort and destroy

Returns
Result code. See *Result Codes* for more details.

ncclResult_t **ncclCommSplit**(*ncclComm_t* comm, int color, int key, *ncclComm_t* *newcomm, *ncclConfig_t* *config)

Create one or more communicators from an existing one.

Creates one or more communicators from an existing one. Ranks with the same color will end up in the same communicator. Within the new communicator, key will be used to order ranks. NCCL_SPLIT_NOCOLOR as color will indicate the rank will not be part of any group and will therefore return a NULL communicator. If config is NULL, the new communicator will inherit the original communicator's configuration

Parameters

- **comm** – **[in]** Original communicator object for this rank
- **color** – **[in]** Color to assign this rank
- **key** – **[in]** Key used to order ranks within the same new communicator
- **newcomm** – **[out]** Pointer to new communicator
- **config** – **[in]** Config file for new communicator. May be NULL to inherit from comm

Returns
Result code. See *Result Codes* for more details.

ncclResult_t **ncclCommShrink**(*ncclComm_t* comm, int *excludeRanksList, int excludeRanksCount, *ncclComm_t* *newcomm, *ncclConfig_t* *config, int shrinkFlags)

Shrink existing communicator.

Ranks in excludeRanksList will be removed form the existing communicator. Within the new communicator, ranks will be re-ordered to fill the gap of removed ones. If config is NULL, the new communicator will inherit the original communicator's configuration. The flag enables NCCL to adapt to various states of the parent communicator, see NCCL_SHRINK flags.

Parameters

- **comm** – **[in]** Original communicator object for this rank
- **excludeRanksList** – **[in]** List of ranks to be exluded
- **excludeRanksCount** – **[in]** Number of ranks to be excluded
- **newcomm** – **[out]** Pointer to new communicator
- **config** – **[in]** Config file for new communicator. May be NULL to inherit from comm
- **shrinkFlags** – **[in]** Flag to adapt to various states of the parent communicator (see NCCL_SHRINK flags)

Returns

Result code. See *Result Codes* for more details.

ncclResult_t **pncclCommShrink**(*ncclComm_t* comm, int *excludeRanksList, int excludeRanksCount, *ncclComm_t* *newcomm, *ncclConfig_t* *config, int shrinkFlags)

ncclResult_t **ncclCommInitRankScalable**(*ncclComm_t* *newcomm, int nranks, int myrank, int nId, *ncclUniqueId* *commIds, *ncclConfig_t* *config)

Creates a new communicator (multi thread/process version), similar to ncclCommInitRankConfig.

Allows to use more than one *ncclUniqueId* (up to one per rank), indicated by nId, to accelerate the init operation. The number of ncclUniqueIds and their order must be the same for every rank.

Parameters

- **newcomm** – **[out]** Pointer to new communicator
- **nranks** – **[in]** Total number of ranks participating in this communicator
- **myrank** – **[in]** Current rank
- **nId** – **[in]** Number of unique IDs
- **commIds** – **[in]** List of unique IDs
- **config** – **[in]** Config file for new communicator. May be NULL to inherit from comm

Returns

Result code. See *Result Codes* for more details.

const char ***ncclGetErrorString**(*ncclResult_t* result)

Returns a string for each result code.

Returns a human-readable string describing the given result code.

Parameters

result – **[in]** Result code to get description for

Returns

String containing description of result code.

const char *ncclGetLastError(*ncclComm_t* comm)

void ncclResetDebugInit()

ncclResult_t ncclCommGetAsyncError(*ncclComm_t* comm, *ncclResult_t* *asyncError)

Checks whether the comm has encountered any asynchronous errors.

Query whether the provided communicator has encountered any asynchronous errors

Parameters

- **comm** – **[in]** Communicator to query
- **asyncError** – **[out]** Pointer to where result code will be stored

Returns

Result code. See *Result Codes* for more details.

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

Gets the number of ranks in the communicator clique.

Returns the number of ranks in the communicator clique (as set during initialization)

Parameters

- **comm** – **[in]** Communicator to query
- **count** – **[out]** Pointer to where number of ranks will be stored

Returns

Result code. See *Result Codes* for more details.

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

Get the ROCm device index associated with a communicator.

Returns the ROCm device number associated with the provided communicator.

Parameters

- **comm** – **[in]** Communicator to query
- **device** – **[out]** Pointer to where the associated ROCm device index will be stored

Returns

Result code. See *Result Codes* for more details.

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

Get the rank associated with a communicator.

Returns the user-ordered "rank" associated with the provided communicator.

Parameters

- **comm** – **[in]** Communicator to query
- **rank** – **[out]** Pointer to where the associated rank will be stored

Returns

Result code. See *Result Codes* for more details.

ncclResult_t ncclCommRegister(const *ncclComm_t* comm, void *buff, size_t size, void **handle)

ncclResult_t ncclCommDeregister(const *ncclComm_t* comm, void *handle)

ncclResult_t **ncclCommWindowRegister**(*ncclComm_t* comm, void *buff, size_t size, *ncclWindow_t* *win, int winFlags)

ncclResult_t **ncclCommWindowDeregister**(*ncclComm_t* comm, *ncclWindow_t* win)

ncclResult_t **ncclRedOpCreatePreMulSum**(*ncclRedOp_t* *op, void *scalar, *ncclDataType_t* datatype, *ncclScalarResidence_t* residence, *ncclComm_t* comm)

Create a custom pre-multiplier reduction operator.

Creates a new reduction operator which pre-multiplies input values by a given scalar locally before reducing them with peer values via summation. For use only with collectives launched against *comm* and *datatype*. The residence* argument indicates how/when the memory pointed to by *scalar* will be dereferenced. Upon return, the newly created operator's handle is stored in *op*.

Parameters

- **op** – **[out]** Pointer to where newly created custom reduction operator is to be stored
- **scalar** – **[in]** Pointer to scalar value.
- **datatype** – **[in]** Scalar value datatype
- **residence** – **[in]** Memory type of the scalar value
- **comm** – **[in]** Communicator to associate with this custom reduction operator

Returns

Result code. See *Result Codes* for more details.

ncclResult_t **ncclRedOpDestroy**(*ncclRedOp_t* op, *ncclComm_t* comm)

Destroy custom reduction operator.

Destroys the reduction operator *op*. The operator must have been created by ncclRedOpCreatePreMul with the matching communicator *comm*. An operator may be destroyed as soon as the last RCCL function which is given that operator returns.

Parameters

- **op** – **[in]** Custom reduction operator is to be destroyed
- **comm** – **[in]** Communicator associated with this reduction operator

Returns

Result code. See *Result Codes* for more details.

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

Reduce.

Reduces data arrays of length *count* in *sendbuff* into *recvbuff* using *op* operation. recvbuff* may be NULL on all calls except for root device. root* is the rank (not the HIP device) where data will reside after the operation is complete. In-place operation will happen if sendbuff == recvbuff.

Parameters

- **sendbuff** – **[in]** Local device data buffer to be reduced
- **recvbuff** – **[out]** Data buffer where result is stored (only for *root* rank). May be null for other ranks.
- **count** – **[in]** Number of elements in every send buffer
- **datatype** – **[in]** Data buffer element datatype
- **op** – **[in]** Reduction operator type

- **root** – **[in]** Rank where result data array will be stored
- **comm** – **[in]** Communicator group object to execute on
- **stream** – **[in]** HIP stream to execute collective on

Returns
Result code. See *Result Codes* for more details.

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

(Deprecated) Broadcast (in-place)

Copies *count* values from *root* to all other devices. root is the rank (not the CUDA device) where data resides before the operation is started. This operation is implicitly in-place.

Parameters

- **buff** – **[inout]** Input array on *root* to be copied to other ranks. Output array for all ranks.
- **count** – **[in]** Number of elements in data buffer
- **datatype** – **[in]** Data buffer element datatype
- **root** – **[in]** Rank owning buffer to be copied to others
- **comm** – **[in]** Communicator group object to execute on
- **stream** – **[in]** HIP stream to execute collective on

Returns
Result code. See *Result Codes* for more details.

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

Broadcast.

Copies *count* values from *sendbuff* on *root* to *recvbuff* on all devices. root* is the rank (not the HIP device) where data resides before the operation is started. sendbuff* may be NULL on ranks other than *root*. In-place operation will happen if *sendbuff* == *recvbuff*.

Parameters

- **sendbuff** – **[in]** Data array to copy (if *root*). May be NULL for other ranks
- **recvbuff** – **[in]** Data array to store received array
- **count** – **[in]** Number of elements in data buffer
- **datatype** – **[in]** Data buffer element datatype
- **root** – **[in]** Rank of broadcast root
- **comm** – **[in]** Communicator group object to execute on
- **stream** – **[in]** HIP stream to execute collective on

Returns
Result code. See *Result Codes* for more details.

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

All-Reduce.

Reduces data arrays of length *count* in *sendbuff* using *op* operation, and leaves identical copies of result on each *recvbuff*. In-place operation will happen if sendbuff == recvbuff.

Parameters

- **sendbuff** – **[in]** Input data array to reduce
- **recvbuff** – **[out]** Data array to store reduced result array
- **count** – **[in]** Number of elements in data buffer
- **datatype** – **[in]** Data buffer element datatype
- **op** – **[in]** Reduction operator
- **comm** – **[in]** Communicator group object to execute on
- **stream** – **[in]** HIP stream to execute collective on

Returns

Result code. See *Result Codes* for more details.

ncclResult_t **ncclAllReduceWithBias**(const void *sendbuff, void *recvbuff, size_t count, *ncclDataType_t* datatype, *ncclRedOp_t* op, *ncclComm_t* comm, hipStream_t stream, const void *acc)

All-Reduce-with-Bias.

Reduces data arrays of length *count* in *sendbuff* using *op* operation, and leaves identical copies of result on each *recvbuff*. In-place operation will happen if sendbuff == recvbuff.

Parameters

- **sendbuff** – **[in]** Input data array to reduce
- **recvbuff** – **[out]** Data array to store reduced result array
- **count** – **[in]** Number of elements in data buffer
- **datatype** – **[in]** Data buffer element datatype
- **op** – **[in]** Reduction operator
- **comm** – **[in]** Communicator group object to execute on
- **stream** – **[in]** HIP stream to execute collective on
- **acc** – **[in]** Bias data array to reduce

Returns

Result code. See *Result Codes* for more details.

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

Reduce-Scatter.

Reduces data in *sendbuff* using *op* operation and leaves reduced result scattered over the devices so that *recvbuff* on rank i will contain the i-th block of the result. Assumes sendcount is equal to nranks*recvcount, which means that *sendbuff* should have a size of at least nranks*recvcount elements. In-place operations will happen if recvbuff == sendbuff + rank * recvcount.

Parameters

- **sendbuff** – **[in]** Input data array to reduce
- **recvbuff** – **[out]** Data array to store reduced result subarray
- **recvcount** – **[in]** Number of elements each rank receives
- **datatype** – **[in]** Data buffer element datatype

- **op** – **[in]** Reduction operator
- **comm** – **[in]** Communicator group object to execute on
- **stream** – **[in]** HIP stream to execute collective on

Returns
Result code. See *Result Codes* for more details.

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

All-Gather.

Each device gathers *sendcount* values from other GPUs into *recvbuff*, receiving data from rank i at offset i*sendcount. Assumes recvcount is equal to nranks*sendcount, which means that recvbuff should have a size of at least nranks*sendcount elements. In-place operations will happen if sendbuff == recvbuff + rank * sendcount.

Parameters

- **sendbuff** – **[in]** Input data array to send
- **recvbuff** – **[out]** Data array to store the gathered result
- **sendcount** – **[in]** Number of elements each rank sends
- **datatype** – **[in]** Data buffer element datatype
- **comm** – **[in]** Communicator group object to execute on
- **stream** – **[in]** HIP stream to execute collective on

Returns
Result code. See *Result Codes* for more details.

ncclResult_t **ncclSend**(const void *sendbuff, size_t count, *ncclDataType_t* datatype, int peer, *ncclComm_t* comm, hipStream_t stream)

Send.

Send data from *sendbuff* to rank *peer*. Rank *peer* needs to call ncclRecv with the same *datatype* and the same *count* as this rank. This operation is blocking for the GPU. If multiple ncclSend and ncclRecv operations need to progress concurrently to complete, they must be fused within a ncclGroupStart / ncclGroupEnd section.

Parameters

- **sendbuff** – **[in]** Data array to send
- **count** – **[in]** Number of elements to send
- **datatype** – **[in]** Data buffer element datatype
- **peer** – **[in]** Peer rank to send to
- **comm** – **[in]** Communicator group object to execute on
- **stream** – **[in]** HIP stream to execute collective on

Returns
Result code. See *Result Codes* for more details.

ncclResult_t **ncclRecv**(void *recvbuff, size_t count, *ncclDataType_t* datatype, int peer, *ncclComm_t* comm, hipStream_t stream)

Receive.

Receive data from rank *peer* into *recvbuff*. Rank *peer* needs to call ncclSend with the same datatype and the same count as this rank. This operation is blocking for the GPU. If multiple ncclSend and ncclRecv operations need to progress concurrently to complete, they must be fused within a ncclGroupStart/ ncclGroupEnd section.

Parameters

- **recvbuff** – **[out]** Data array to receive
- **count** – **[in]** Number of elements to receive
- **datatype** – **[in]** Data buffer element datatype
- **peer** – **[in]** Peer rank to send to
- **comm** – **[in]** Communicator group object to execute on
- **stream** – **[in]** HIP stream to execute collective on

Returns

Result code. See *Result Codes* for more details.

ncclResult_t **ncclGather**(const void *sendbuff, void *recvbuff, size_t sendcount, *ncclDataType_t* datatype, int root, *ncclComm_t* comm, hipStream_t stream)

Gather.

Root device gathers *sendcount* values from other GPUs into *recvbuff*, receiving data from rank i at offset i*sendcount. Assumes recvcount is equal to nranks*sendcount, which means that *recvbuff* should have a size of at least nranks*sendcount elements. In-place operations will happen if sendbuff == recvbuff + rank * sendcount. recvbuff* may be NULL on ranks other than *root*.

Parameters

- **sendbuff** – **[in]** Data array to send
- **recvbuff** – **[out]** Data array to receive into on *root*.
- **sendcount** – **[in]** Number of elements to send per rank
- **datatype** – **[in]** Data buffer element datatype
- **root** – **[in]** Rank that receives data from all other ranks
- **comm** – **[in]** Communicator group object to execute on
- **stream** – **[in]** HIP stream to execute collective on

Returns

Result code. See *Result Codes* for more details.

ncclResult_t **ncclScatter**(const void *sendbuff, void *recvbuff, size_t recvcount, *ncclDataType_t* datatype, int root, *ncclComm_t* comm, hipStream_t stream)

Scatter.

Scattered over the devices so that recvbuff on rank i will contain the i-th block of the data on root. Assumes sendcount is equal to nranks*recvcount, which means that *sendbuff* should have a size of at least nranks*recvcount elements. In-place operations will happen if recvbuff == sendbuff + rank * recvcount.

Parameters

- **sendbuff** – **[in]** Data array to send (on *root* rank). May be NULL on other ranks.
- **recvbuff** – **[out]** Data array to receive partial subarray into
- **recvcount** – **[in]** Number of elements to receive per rank

- **datatype** – **[in]** Data buffer element datatype
- **root** – **[in]** Rank that scatters data to all other ranks
- **comm** – **[in]** Communicator group object to execute on
- **stream** – **[in]** HIP stream to execute collective on

Returns
Result code. See *Result Codes* for more details.

ncclResult_t **ncclAllToAll**(const void *sendbuff, void *recvbuff, size_t count, *ncclDataType_t* datatype, *ncclComm_t* comm, hipStream_t stream)

All-To-All.

Device (i) send (j)th block of data to device (j) and be placed as (i)th block. Each block for sending/receiving has *count* elements, which means that *recvbuff* and *sendbuff* should have a size of nranks*count elements. In-place operation is NOT supported. It is the user's responsibility to ensure that sendbuff and recvbuff are distinct.

Parameters

- **sendbuff** – **[in]** Data array to send (contains blocks for each other rank)
- **recvbuff** – **[out]** Data array to receive (contains blocks from each other rank)
- **count** – **[in]** Number of elements to send between each pair of ranks
- **datatype** – **[in]** Data buffer element datatype
- **comm** – **[in]** Communicator group object to execute on
- **stream** – **[in]** HIP stream to execute collective on

Returns
Result code. See *Result Codes* for more details.

ncclResult_t **ncclAllToAllv**(const void *sendbuff, const size_t sendcounts[], const size_t sdispls[], void *recvbuff, const size_t recvcounts[], const size_t rdispls[], *ncclDataType_t* datatype, *ncclComm_t* comm, hipStream_t stream)

All-To-Allv.

Device (i) sends sendcounts[j] of data from offset sdispls[j] to device (j). At the same time, device (i) receives recvcounts[j] of data from device (j) to be placed at rdispls[j]. sendcounts, sdispls, recvcounts and rdispls are all measured in the units of datatype, not bytes. In-place operation will happen if sendbuff == recvbuff.

Parameters

- **sendbuff** – **[in]** Data array to send (contains blocks for each other rank)
- **sendcounts** – **[in]** Array containing number of elements to send to each participating rank
- **sdispls** – **[in]** Array of offsets into *sendbuff* for each participating rank
- **recvbuff** – **[out]** Data array to receive (contains blocks from each other rank)
- **recvcounts** – **[in]** Array containing number of elements to receive from each participating rank
- **rdispls** – **[in]** Array of offsets into *recvbuff* for each participating rank
- **datatype** – **[in]** Data buffer element datatype
- **comm** – **[in]** Communicator group object to execute on

- **stream** – **[in]** HIP stream to execute collective on

Returns
Result code. See *Result Codes* for more details.

ncclResult_t **mscclLoadAlgo**(const char *mscclAlgoFilePath, *mscclAlgoHandle_t* *mscclAlgoHandle, int rank)

MSCCL Load Algorithm.

Deprecated:

This function has been removed from the public API.

Load MSCCL algorithm file specified in mscclAlgoFilePath and return its handle via mscclAlgoHandle. This API is expected to be called by MSCCL scheduler instead of end users.

Parameters

- **mscclAlgoFilePath** – **[in]** Path to MSCCL algorithm file
- **mscclAlgoHandle** – **[out]** Returned handle to MSCCL algorithm
- **rank** – **[in]** Current rank

Returns
Result code. See *Result Codes* for more details.

ncclResult_t **mscclRunAlgo**(const void *sendBuff, const size_t sendCounts[], const size_t sDisPls[], void *recvBuff, const size_t recvCounts[], const size_t rDisPls[], size_t count, *ncclDataType_t* dataType, int root, int peer, *ncclRedOp_t* op, *mscclAlgoHandle_t* mscclAlgoHandle, *ncclComm_t* comm, hipStream_t stream)

MSCCL Run Algorithm.

Deprecated:

This function has been removed from the public API.

Run MSCCL algorithm specified by mscclAlgoHandle. The parameter list merges all possible parameters required by different operations as this is a general-purposed API. This API is expected to be called by MSCCL scheduler instead of end users.

Parameters

- **sendBuff** – **[in]** Data array to send
- **sendCounts** – **[in]** Array containing number of elements to send to each participating rank
- **sDisPls** – **[in]** Array of offsets into *sendbuff* for each participating rank
- **recvBuff** – **[out]** Data array to receive
- **recvCounts** – **[in]** Array containing number of elements to receive from each participating rank
- **rDisPls** – **[in]** Array of offsets into *recvbuff* for each participating rank
- **count** – **[in]** Number of elements
- **dataType** – **[in]** Data buffer element datatype
- **root** – **[in]** Root rank index
- **peer** – **[in]** Peer rank index

- **op** – **[in]** Reduction operator
- **mscclAlgoHandle** – **[in]** Handle to MSCCL algorithm
- **comm** – **[in]** Communicator group object to execute on
- **stream** – **[in]** HIP stream to execute collective on

Returns
Result code. See *Result Codes* for more details.

ncclResult_t **mscclUnloadAlgo**(*mscclAlgoHandle_t* mscclAlgoHandle)

MSCCL Unload Algorithm.

Deprecated:

This function has been removed from the public API.

Unload MSCCL algorithm previous loaded using its handle. This API is expected to be called by MSCCL scheduler instead of end users.

Parameters
mscclAlgoHandle – **[in]** Handle to MSCCL algorithm to unload

Returns
Result code. See *Result Codes* for more details.

ncclResult_t **ncclGroupStart**()

Group Start.

Start a group call. All calls to RCCL until ncclGroupEnd will be fused into a single RCCL operation. Nothing will be started on the HIP stream until ncclGroupEnd.

Returns
Result code. See *Result Codes* for more details.

ncclResult_t **ncclGroupEnd**()

Group End.

End a group call. Start a fused RCCL operation consisting of all calls since ncclGroupStart. Operations on the HIP stream depending on the RCCL operations need to be called after ncclGroupEnd.

Returns
Result code. See *Result Codes* for more details.

ncclResult_t **ncclGroupSimulateEnd**(*ncclSimInfo_t* *simInfo)

ncclResult_t **pncclGroupSimulateEnd**(*ncclSimInfo_t* *simInfo)

group **rccl_result_code**

The various result codes that RCCL API calls may return

Enums

enum **ncclResult_t**

Result type.

Return codes aside from ncclSuccess indicate that a call has failed

Values:

enumerator **ncclSuccess**

No error

enumerator **ncclUnhandledCudaError**

Unhandled HIP error

enumerator **ncclSystemError**

Unhandled system error

enumerator **ncclInternalError**

Internal Error - Please report to RCCL developers

enumerator **ncclInvalidArgument**

Invalid argument

enumerator **ncclInvalidUsage**

Invalid usage

enumerator **ncclRemoteError**

Remote process exited or there was a network error

enumerator **ncclInProgress**

RCCL operation in progress

enumerator **ncclNumResults**

Number of result types

group **rccl_config_type**

Structure that allows for customizing Communicator behavior via ncclCommInitRankConfig

Defines

NCCL_CONFIG_INITIALIZER

group rccl_api_version

API call that returns RCCL version

group rccl_api_communicator

API calls that operate on communicators. Communicators objects are used to launch collective communication operations. Unique ranks between 0 and N-1 must be assigned to each HIP device participating in the same Communicator. Using the same HIP device for multiple ranks of the same Communicator is not supported at this time.

group rccl_api_errcheck

API calls that check for errors

group rccl_api_comminfo

API calls that query communicator information

group rccl_api_enumerations

Enumerations used by collective communication calls

Enums

enum ncclRedOp_dummy_t

Dummy reduction enumeration.

Dummy reduction enumeration used to determine value for ncclMaxRedOp

Values:

enumerator ncclNumOps_dummy

enum ncclRedOp_t

Reduction operation selector.

Enumeration used to specify the various reduction operations ncclNumOps is the number of built-in ncclRedOp_t values and serves as the least possible value for dynamic ncclRedOp_t values constructed by ncclRedOpCreate functions.

ncclMaxRedOp is the largest valid value for ncclRedOp_t and is defined to be the largest signed value (since compilers are permitted to use signed enums) that won't grow sizeof(ncclRedOp_t) when compared to previous RCCL versions to maintain ABI compatibility.

Values:

enumerator ncclSum

Sum

enumerator **ncclProd**

Product

enumerator **ncclMax**

Max

enumerator **ncclMin**

Min

enumerator **ncclAvg**

Average

enumerator **ncclNumOps**

Number of built-in reduction ops

enumerator **ncclMaxRedOp**

Largest value for ncclRedOp_t

enum **ncclDataType_t**

Data types.

Enumeration of the various supported datatype

Values:

enumerator **ncclInt8**

enumerator **ncclChar**

enumerator **ncclUint8**

enumerator **ncclInt32**

enumerator **ncclInt**

enumerator **ncclUint32**

enumerator **ncclInt64**

enumerator **ncclUint64**

enumerator **ncclFloat16**

enumerator **ncclHalf**

enumerator **ncclFloat32**

enumerator **ncclFloat**

enumerator **ncclFloat64**

enumerator **ncclDouble**

enumerator **ncclBfloat16**

enumerator **ncclFloat8e4m3**

enumerator **ncclFloat8e5m2**

enumerator **ncclNumTypes**

group **rccl_api_custom_redop**

API calls relating to creation/destroying custom reduction operator that pre-multiplies local source arrays prior to reduction

Enums

enum **ncclScalarResidence_t**

Location and dereferencing logic for scalar arguments.

Enumeration specifying memory location of the scalar argument. Based on where the value is stored, the argument will be dereferenced either while the collective is running (if in device memory), or before the ncclRedOpCreate() function returns (if in host memory).

Values:

enumerator **ncclScalarDevice**

Scalar is in device-visible memory

enumerator **ncclScalarHostImmediate**

Scalar is in host-visible memory

group **rccl_collective_api**

Collective communication operations must be called separately for each communicator in a communicator clique.

They return when operations have been enqueued on the HIP stream. Since they may perform inter-CPU synchronization, each call has to be done from a different thread or process, or need to use Group Semantics (see below).

group **msccl_api**

API calls relating to the optional MSCCL algorithm datapath

Typedefs

typedef int **mscclAlgoHandle_t**

Opaque handle to MSCCL algorithm.

group **rccl_group_api**

When managing multiple GPUs from a single thread, and since RCCL collective calls may perform inter-CPU synchronization, we need to "group" calls for different ranks/devices into a single call.

Grouping RCCL calls as being part of the same collective operation is done using ncclGroupStart and ncclGroupEnd. ncclGroupStart will enqueue all collective calls until the ncclGroupEnd call, which will wait for all calls to be complete. Note that for collective communication, ncclGroupEnd only guarantees that the operations are enqueued on the streams, not that the operation is effectively done.

Both collective communication and ncclCommInitRank can be used in conjunction of ncclGroupStart/ncclGroupEnd, but not together.

Group semantics also allow to fuse multiple operations on the same device to improve performance (for aggregated collective calls), or to permit concurrent progress of multiple send/receive operations.

page **deprecated**

Global *mscclLoadAlgo* (const char *mscclAlgoFilePath, mscclAlgoHandle_t *mscclAlgoHandle, int rank)

This function has been removed from the public API.

Global *mscclRunAlgo* (const void *sendBuff, const size_t sendCounts[], const size_t sDisPls[], void *recvBuff, const size_t recvCounts[], const size_t rDisPls[], size_t count, ncclDataType_t dataType, int root, int peer, ncclRedOp_t op, mscclAlgoHandle_t mscclAlgoHandle, ncclComm_t comm, hipStream_t stream)

This function has been removed from the public API.

Global *mscclUnloadAlgo* (mscclAlgoHandle_t mscclAlgoHandle)

This function has been removed from the public API.

dir **src**

page **index**

10.1 Introduction

RCCL (pronounced "Rickle") is a stand-alone library of standard collective communication routines for GPUs, implementing all-reduce, all-gather, reduce, broadcast, reduce-scatter, gather, scatter, and all-to-all. There is also initial support for direct GPU-to-GPU send and receive operations. It has been optimized to achieve high bandwidth on platforms using PCIe, xGMI as well as networking using InfiniBand Verbs or TCP/IP sockets. RCCL supports an arbitrary number of GPUs installed in a single node or multiple nodes, and can be used in either single- or multi-process (e.g., MPI) applications.

The collective operations are implemented using ring and tree algorithms and have been optimized for throughput and latency. For best performance, small operations can be either batched into larger operations or aggregated through the API.

10.2 RCCL API Contents

10.3 RCCL API File

CHAPTER

ELEVEN

RCCL ENVIRONMENT VARIABLES

This section describes the most important RCCL environment variables, which are grouped by functionality.

11.1 Configuration and setup

The configuration and setup environment variables for RCCL are collected in the following table.

Environment variable	Value
`NCCL_CONF_FILE` Specifies the path to the RCCL configuration file.	String path to configuration file Default: `~/.rccl.conf` or `/etc/rccl.conf`
`NCCL_HOSTID` Sets the host identifier for multi-node communication.	String value for host identification Used for host hash generation

11.2 Logging and debugging

The logging and debugging environment variables for RCCL are collected in the following table.

Environment variable	Value
`RCCL_LOG_LEVEL` Controls RCCL logging verbosity.	Integer value (default: 1) Higher values increase logging detail
`NCCL_DEBUG_SUBSYS` Controls which subsystems generate debug output.	Comma-separated list of subsystems (e.g., `INIT,COLL`) Prefix with `^` to invert selection

11.3 Algorithm and protocol control

The algorithm and protocol control environment variables for RCCL are collected in the following table.

Environment variable	Value
`NCCL_ALGO` Forces specific algorithm selection for collectives.	Algorithm name string Used to override automatic algorithm selection
`NCCL_PROTO` Forces specific protocol selection for communication.	Protocol name string Used to override automatic protocol selection

11.4 Network and topology

The network and topology environment variables for RCCL are collected in the following table.

Environment variable	Value
NCCL_IB_HCA Specifies InfiniBand device:port to use.	Device specification string Prefix with ^ for exclusion, = for exact match
NCCL_IB_GID_INDEX Defines the Global ID index used in RoCE mode.	Integer value (default: -1) See InfiniBand show_gids command for valid values
NCCL_SOCKET_IFNAME Specifies which IP interfaces to use for communication.	Interface prefix string or list Multiple prefixes separated by , Prefix with ^ for exclusion, = for exact match Example: eth (all eth interfaces), =eth0 (exact match)
NCCL_SOCKET_FAMILY Forces IPv4/IPv6 interface selection.	AF_INET: Force IPv4 AF_INET6: Force IPv6 Unset: Use first available
NCCL_NET_MERGE_LEVEL Controls network device merging behavior.	Integer value specifying merge level Default: PATH_PORT
NCCL_NET_FORCE_MERGE Forces merging of network devices.	String specifying forced merge configuration
NCCL_RINGS Defines custom ring topology.	Ring topology specification string Overrides automatic topology detection
RCCL_TREES Defines custom tree topology.	Tree topology specification string Alternative to ring topology
NCCL_RINGS_REMAP Controls ring remapping for specific topologies.	Remapping specification string Used with Rome 4P2H topology

11.5 Development and testing (advanced)

The development and testing environment variables for RCCL are collected in the following table. These variables are primarily intended for debugging and development purposes.

Environment variable	Value
`CUDA_LAUNCH_BLOCKING` Controls CUDA kernel launch blocking behavior.	`0`: Non-blocking launches 1 or non-zero: Blocking launches
`NCCL_COMM_ID` Enables multi-process mode in test applications.	Any non-empty value enables multi-process mode Used with test executables for distributed testing

CHAPTER

TWELVE

LICENSE

Attributions

Contains contributions from NVIDIA.

The U.S. Department of Energy funded the development of this software under subcontract 7078610 with Lawrence Berkeley National Laboratory.

This code also includes files from the NVIDIA Tools Extension SDK project.

See:

https://github.com/NVIDIA/NVTX

for more information and license details.

CHAPTER

THIRTEEN

ATTRIBUTIONS

Contains contributions from NVIDIA.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of NVIDIA CORPORATION, Lawrence Berkeley National Laboratory, the U.S. Department of Energy, nor the names of their contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS ‘’AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The U.S. Department of Energy funded the development of this software under subcontract 7078610 with Lawrence Berkeley National Laboratory.

This code also includes files from the NVIDIA Tools Extension SDK project.

For more information and license details, see https://github.com/NVIDIA/NVTX

INDEX

P

R