Intel® oneAPI Collective Communications Library

Intel® oneAPI Collective Communications Library (oneCCL) provides an efficient implementation of communication patterns used in deep learning.

oneCCL features include:

- Built on top of lower-level communication middleware – Intel® MPI Library and libfabrics.
- Optimized to drive scalability of communication patterns by allowing to easily trade-off compute for communication performance.
- Enables a set of DL-specific optimizations, such as prioritization, persistent operations, or out-of-order execution.
- Works across various interconnects: Intel(R) Omni-Path Architecture, InfiniBand*, and Ethernet.
- Provides common API sufficient to support communication workflows within Deep Learning frameworks (such as Caffe*, nGraph*, or Horovod*).

oneCCL package comprises the oneCCL Software Development Kit (SDK) and the Intel(R) MPI Library Runtime components.

Contents:

Get Started

Programming Model

- CPU support

General Configuration

- Execution of collective operations
- Transport selection

Advanced Configuration

- Selection of collective algorithms
- Caching of collective operations
- Prioritization of collective operations
- Fusion of collective operations
- Sparse collective operations
- Unordered collectives support
- Fault tolerance / elasticity

Reference Materials

- Environment variables
- Library API

Prerequisites

Before you start using oneCCL, make sure to set up the library environment. With oneCCL installed into `<installdir>`, there are two ways to set up the environment:

- Using standalone oneCCL package:

  ```
  $ source <installdir>/setvars.sh
  ```

- Using oneCCL from Intel® oneAPI Base Toolkit:

  ```
  $ sourse <installdir>/setvars.sh
  ```

  By default, `<installdir>` is `/opt/intel/inteloneapi`.

Installation

This page explains how to install and configure the Intel® oneAPI Collective Communications Library (oneCCL).

oneCCL supports different installation scenarios:

- Installation using command line interface
- Installation using tar.gz
- Installation using RPM

> **Note**
>
> Visit Intel® oneAPI Collective Communications Library System Requirements to learn about hardware and software requirements for oneCCL.

Installation using Command Line Interface

To install oneCCL using command line interface (CLI), follow these steps:

1. Go to the `ccl` folder:

   ```
   cd ccl
   ```

2. Create a new folder:

   ```
   mkdir build
   ```

3. Go to the folder created:

   ```
   cd build
   ```

4. Launch CMake:

```
cmake ..
```

5. Install the product:

   ```
   make -j install
   ```

In order to have a clear build, create a new `build` directory and invoke `cmake` within the directory.

Custom Installation

You can customize CLI-based installation (for example, specify directory, compiler, and build type):

- To speciify **installation directory**, modify the `cmake` command:

  ```
  cmake .. -DCMAKE_INSTALL_PREFIX=/path/to/installation/directory
  ```

  If no `-DCMAKE_INSTALL_PREFIX` is specified, oneCCL is installed into the `_install` subdirectory of the current build directory. For example, `ccl/build/_install`.
- To specify **compiler**, modify the `cmake` command:

  ```
  cmake .. -DCMAKE_C_COMPILER=your_c_compiler -DCMAKE_CXX_COMPILER=your_cxx_compiler
  ```

  If `CMAKE_CXX_COMPILER` requires `SYCL` cross-platform abstraction level it should be specified in `-DCOMPUTE_RUNTIME` (`compute++` and `dpcpp` supported only):

  ```
  cmake .. -DCMAKE_C_COMPILER=your_c_compiler -DCMAKE_CXX_COMPILER=compute++ -
  DCOMPUTE_RUNTIME=computecpp
  cmake .. -DCMAKE_C_COMPILER=your_c_compiler -DCMAKE_CXX_COMPILER=dpcpp -
  DCOMPUTE_RUNTIME=dpcpp
  ```

  OpenCL search location path hint can be specified by using standart environment `OPENCLROOT` additionally:

```
OPENCLROOT=your_opencl_location cmake .. -DCMAKE_C_COMPILER=your_c_compiler -
DCMAKE_CXX_COMPILER=compute++ -DCOMPUTE_RUNTIME=computecpp
```

- To specify the **build type**, modify the `cmake` command:

  ```
  cmake .. -DCMAKE_BUILD_TYPE=[Debug|Release|RelWithDebInfo|MinSizeRel]
  ```

- To enable `make` verbose output to see all parameters used by `make` during compilation and linkage, modify the `make` command as follows:

  ```
  make -j VERBOSE=1
  ```

- To archive installed files:

  ```
  make -j install
  ```

- To build with Address Sanitizer, modify the `cmake` command as follow:

  ```
  cmake .. -DCMAKE_BUILD_TYPE=Debug -DWITH_ASAN=true
  ```

  Make sure that `libasan.so` exists.

  Note

  Address sanitizer only works in the debug build.

Binary releases are available on our release page.

Installation using tar.gz

To install oneCCL using the tar.gz file in a user mode, execute the following commands:

```
$ tar zxf l_ccl-devel-64-<version>.<update>.<package#>.tgz
$ cd l_ccl_<version>.<update>.<package#>
$ ./install.sh
```

There is no uninstall script. To uninstall oneCCL, delete the whole installation directory.

Installation using RPM

You can get oneCCL through the RPM Package Manager. To install the library in a root mode using RPM, follow these steps:

1. Log in as root.
2. Install the following package:

```
$ rpm -i intel-ccl-devel-64-<version>.<update>-<package#>.x86_64.rpm
$
$ where ``<version>.<update>-<package#>`` is a string. For example, ``2017.0-009``.
```

To uninstall oneCCL using the RPM Package Manager, execute this command:

```
$ rpm -e intel-ccl-devel-64-<version>.<update>-<package#>.x86_64
```

Sample application

The sample code below shows how to use oneCCL API to perform allreduce communication for SYCL* buffers:

```
#include <iostream>
#include <stdio.h>

#include <CL/sycl.hpp>
#include "ccl.h"

#define COUNT     (10 * 1024 * 1024)
#define COLL_ROOT (0)

using namespace std;
using namespace cl::sycl;
using namespace cl::sycl::access;

int main(int argc, char** argv)
{
    int i = 0;
    size_t size = 0;
    size_t rank = 0;

    cl::sycl::queue q;
    cl::sycl::buffer<int, 1> sendbuf(COUNT);
    cl::sycl::buffer<int, 1> recvbuf(COUNT);
    ccl_request_t request;
    ccl_stream_t stream;

    ccl_init();

    ccl_get_comm_rank(NULL, &rank);
    ccl_get_comm_size(NULL, &size);

    // create CCL stream based on SYCL* command queue
    ccl_stream_create(ccl_stream_sycl, &q, &stream);

    {
            /* open buffers and initialize them on the CPU side */
            auto host_acc_sbuf = sendbuf.get_access<mode::write>();
            auto host_acc_rbuf = recvbuf.get_access<mode::write>();
            for (i = 0; i < COUNT; i++) {
                host_acc_sbuf[i] = rank;
                host_acc_rbuf[i] = -1;
            }
    }

    /* open sendbuf and modify it on the target device side */
    q.submit([&](cl::sycl::handler& cgh) {
       auto dev_acc_sbuf = sendbuf.get_access<mode::write>(cgh);
       cgh.parallel_for<class allreduce_test_sbuf_modify>(range<1>{COUNT}, [=](item<1> id)
{
           dev_acc_sbuf[id] += 1;
       });
    });

    /* invoke ccl_allreduce on the CPU side */
    ccl_allreduce(&sendbuf,
                  &recvbuf,
                  COUNT,
                  ccl_dtype_int,
                  ccl_reduction_sum,
                  NULL,
                  NULL,
                  stream,
                  &request);
    ccl_wait(request);

    /* open recvbuf and check its correctness on the target device side */
    q.submit([&](handler& cgh) {
       auto dev_acc_rbuf = recvbuf.get_access<mode::write>(cgh);
       cgh.parallel_for<class allreduce_test_rbuf_check>(range<1>{COUNT}, [=](item<1> id)
{
           if (dev_acc_rbuf[id] != size * (size + 1) / 2) {
               dev_acc_rbuf[id] = -1;
           }
       });
    });

    /* print out the result of the test on the CPU side */
    if (rank == COLL_ROOT) {
        auto host_acc_rbuf_new = recvbuf.get_access<mode::read>();
        for (i = 0; i < COUNT; i++) {
            if (host_acc_rbuf_new[i] == -1) {
                cout << "FAILED" << endl;
                break;
            }
        }
        if (i == COUNT) {
            cout << "PASSED" << endl;
        }
    }

    ccl_stream_free(stream);

    ccl_finalize();

    return 0;
}
```

Build details

1. oneCCL should be built with SYCL* support.
2. Set up the library environment (see |prerequisites|).
3. Use `clang++` compiler to build the sample:

```
clang++ -I${CCL_ROOT}/include -L${CCL_ROOT}/lib/ -lsycl -lccl -o
ccl_sample ccl_sample.cpp
```

Run the sample

Intel® MPI Library is required for running the sample. Make sure that MPI environment is set up.

To run the sample, use the following command:

```
mpiexec <parameters> ./ccl_sample
```

whereti `<parameters>`ti representstioptionaltimpiexectiparameterstisuchtiastinodeticount,tiprocessestiper node,tihosts,tiandti sotion.

oneCCL Concepts

Intel® oneAPI Collective Communications Library introduces the following list of concepts:

- oneCCL Environment
- oneCCL Stream
- oneCCL Communicator

oneCCL Environment

oneCCL Environment is a singleton object that is used as an entry point into oneCCL. It is defined only for C++ version of API. oneCCL Environment exposes a number of helper methods to manage other CCL objects, such as streams or communicators.

oneCCL Stream

C API

C++ API

```
class stream;
using stream_t = std::unique_ptr<ccl::stream>;
```

CCL Stream encapsulates execution context for communication primitives declared by oneCCL specification. It is an opaque handle that is managed by oneCCL API:

C API

C++ API

```
class environment
{
public:
...
    /**
    * Creates a new ccl stream of @c type with @c native stream
    * @param type the @c ccl::stream_type and may be @c cpu or @c sycl (if 
configured)
    * @param native_stream the existing handle of stream
    */
    stream_t create_stream(ccl::stream_type type = ccl::stream_type::cpu, void*
native_stream = nullptr) const;
}
```

When you create a oneCCL stream object using the API described above, you need to specify the stream type and pass the pointer to the underlying command queue object. For example, for oneAPI device you should pass `ccl::stream_type::sycl` and `cl::sycl::queue` objects.

oneCCL Communicator

C API | C++ API

```
class communicator;
using communicator_t = std::unique_ptr<ccl::communicator>;
```

oneCCL Communicator defines participants of collective communication operations. It is an opaque handle that is managed by oneCCL API:

C API | C++ API

```
class environment
{
public:
...
    /**
    * Creates a new communicator according to @c attr parameters
    * or creates a copy of global communicator, if @c attr is @c nullptr(default)
    * @param attr
    */
    communicator_t create_communicator(const ccl::comm_attr* attr = nullptr) const;
}
```

When you create a oneCCL Communicator, you can optionally specify attributes that control the runtime behaviour of oneCCL implementation.

oneCCL Communicator Attributes

```
typedef struct
{
    /**
    * Used to split global communicator into parts. Ranks with identical color
    * will form a new communicator.
    */
    int color;
} ccl_comm_attr_t;
```

`ccl_comm_attr_t` (`ccl::comm_attr` in C++ version of API) is an extendable structure that serves as a modificator of communicator behaviour.

oneCCL Collective Communication

This section covers collective communcation operations implemented in Intel® oneAPI Collective Communications Library.

- Collective Opertations
- Data Types
- Collective Call Attributes
- Track Communication Progress

Collective Opertations

Intel® oneAPI Collective Communications Library introduces the following list of communication primitives:

- Allgatherv
- Allreduce
- Alltoall
- Alltoallv
- Barrier
- Broadcast
- Reduce

These operations are collective, meaning that all participants of a oneCCL communicator should make a call.

Allgatherv

Allgatherv is a collective communication operation that collects data from all processes within a oneCCL communicator. Each participant gets the same result data. Different participants can contribute segments of different sizes.

C API   C++ API

```
template<class buffer_type>
coll_request_t communicator::allgatherv(
                            const buffer_type* send_buf,
                            size_t send_count,
                            buffer_type* recv_buf,
                            const size_t* recv_counts,
                            const ccl::coll_attr* attr,
                            const ccl::stream_t& stream);
```

send_buf

the buffer with `count` elements of type `buffer_type` that stores local data to be sent

recv_buf [out]

the buffer to store received data, must have the same dimension as `send_buf`

count

the number of elements of type `buffer_type` to be sent by a participant of a oneCCL communicator

recv_counts

the number of elements of type `buffer_type` to be received from each participant of a oneCCL communicator

dtype

datatype of the elements (for C++ API it is inferred from the buffer type)

attr

optional attributes that customize operation

comm

oneCCL communicator for the operation

stream

oneCCL stream associated with the operation

req

object that can be used to track the progress of the operation (returned value for C++ API)

Allreduce

Allreduce includes global reduction operations such as sum, max, min, or user-defined functions, where the result is returned to all members of oneCCL communicator.

C API C++ API

```
template<class buffer_type>
coll_request_t communicator::allreduce(
                        const buffer_type* send_buf,
                        buffer_type* recv_buf,
                        size_t count,
                        ccl::reduction reduction,
                        const ccl::coll_attr* attr,
                        const ccl::stream_t& stream);
```

send_buf

the buffer with `count` elements of `buffer_type` that stores local data to be reduced

recv_buf [out]

the buffer to store reduced result, must have the same dimension as `send_buf`

count

the number of elements of `buffer_type` in `send_buf`

dtype

datatype of the elements (for C++ API it is inferred from the buffer type)

reduction

type of reduction operation to be applied

attr

optional attributes that customize operation

comm

oneCCL communicator for the operation

stream

oneCCL stream associated with the operation

req

object that can be used to track the progress of the operation (returned value for C++ API)

Alltoall

Alltoall is a collective operation in which all processes send the same amount of data to each other and receive the same amount of data from each other. The j-th block sent from the i-th process is received by the j-th process and is placed in the i-th block of `recvbuf`.

C API

C++ API

```
template<class buffer_type>
coll_request_t communicator::alltoall(
                                    const buffer_type* send_buf,
                                    buffer_type* recv_buf,
                                    size_t count,
                                    const ccl::coll_attr* attr,
                                    const ccl::stream_t& stream);
```

send_buf

the buffer with `count` elements of `buffer_type` that stores local data to be sent

recv_buf [out]

the buffer to store received data, must have the same dimension as `send_buf`

count

the number of elements of type `buffer_type` to be sent to or received from each participant of oneCCL communicator

dtype

datatype of the elements (for C++ API it is inferred from the buffer type)

attr

optional attributes that customize operation

comm

oneCCL communicator for the operation

stream

oneCCL stream associated with the operation

req

object that can be used to track the progress of the operation (returned value for C++ API)

Alltoallv

Alltoallv is a generalized version of Alltoall. Alltoallv adds flexibility by allowing a varying amount of data from each process.

C API

C++ API

```
template<class buffer_type>
coll_request_t communicator::alltoallv(
                        const buffer_type* send_buf,
                        const size_t* send_counts,
                        buffer_type* recv_buf,
                        const size_t* recv_counts,
                        const ccl::coll_attr* attr,
                        const ccl::stream_t& stream);
```

send_buf

the buffer with elements of `buffer_type` that stores local data to be sent to all participants

send_counts

the number of elements of type `buffer_type` to be sent to each participant

recv_buf [out]

the buffer to store received data from all participants

recv_counts

the number of elements of type `buffer_type` to be received from each participant

dtype

datatype of the elements (for C++ API it is inferred from the buffer type)

attr

optional attributes that customize operation

comm

oneCCL communicator for the operation

stream

oneCCL stream associated with the operation

req

object that can be used to track the progress of the operation (returned value for C++ API)

Barrier

Blocking barrier synchronization across all members of oneCCL communicator.

C API   **C++ API**

```
void communicator::barrier(const ccl::stream_t& stream);
```

comm

oneCCL communicator for the operation

stream

oneCCL stream associated with the operation

Broadcast

Collective communication operation that broadcasts data from one participant of oneCCL communicator (denoted as root) to all other participants.

C API   **C++ API**

```
template<class buffer_type>
col_request_t communicator::bcast(
                        buffer_type* buf,
                        size_t count,
                        size_t root,
                        const ccl::coll_attr* attr,
                        const ccl::stream_t& stream);
```

buf

serves as send buffer for root and as receive buffer for other participants

count

the number of elements of type `buffer_type` in `send_buf`

dtype

datatype of the elements (for C++ API it is inferred from the buffer type)

root

the rank of the process that broadcasts the data

attr

optional attributes that customize the operation

comm

oneCCL communicator for the operation

stream

oneCCL stream associated with the operation

req

object that can be used to track the progress of the operation (returned value for C++ API)

Reduce

Reduce includes global reduction operations such as sum, max, min, or user-defined functions, where the result is returned to a single member of oneCCL communicator (root).

C API | C++ API

```
template<class buffer_type>
coll_request_t communicator::reduce(
                                 const buffer_type* send_buf,
                                 buffer_type* recv_buf,
                                 size_t count,
                                 ccl::reduction reduction,
                                 size_t root,
                                 const ccl::coll_attr* attr,
                                 const ccl::stream_t& stream);
```

send_buf

the buffer with `count` elements of `buffer_type` that stores local data to be reduced

recv_buf [out]

the buffer to store reduced result, must have the same dimension as `send_buf`

count

the number of elements of `buffer_type` in `send_buf`

dtype

datatype of the elements (for C++ API it is inferred from the buffer type)

reduction

type of reduction operation to be applied

root

the rank of the process that gets the result of reduction

attr

optional attributes that customize operation

comm

oneCCL communicator for the operation

stream

oneCCL stream associated with the operation

req

object that can be used to track the progress of the operation (returned value for C++ API)

The following reduction operations are supported for Allreduce and Reduce primitives:

ccl_reduction_sum

elementwise summation

ccl_reduction_prod

elementwise multiplication

ccl_reduction_min

elementwise min

ccl_reduction_max

elementwise max

ccl_reduction_custom:

class of user-defined operations

Data Types

oneCCL specification defines the following data types that can be used for collective communication operations:

C API C++ API

```
enum datatype: int
{
    dt_char = ccl_dtype_char,
    dt_int = ccl_dtype_int,
    dt_bfp16 = ccl_dtype_bfp16,
    dt_float = ccl_dtype_float,
    dt_double = ccl_dtype_double,
    dt_int64 = ccl_dtype_int64,
    dt_uint64 = ccl_dtype_uint64,
};
```

ccl_dtype_char

Corresponds to *char* in C language

ccl_dtype_int

Corresponds to *signed int* in C language

ccl_dtype_bfp16

BFloat16 datatype

ccl_dtype_float

Corresponds to *float* in C language

ccl_dtype_double

Corresponds to *double* in C language

ccl_dtype_int64

Corresponds to *int64_t* in C language

ccl_dtype_uint64

Corresponds to *uint64_t* in C language

Collective Call Attributes

```
/* Extendable list of collective attributes */
typedef struct
{
    /**
    * Callbacks into application
    * for pre-/post-processing
    * and custom reduction oper
    */
    ccl_prologue_fn_t prologue_fn;
    ccl_epilogue_fn_t epilogue_fn;
    ccl_reduction_fn_t reduction_fn;
    /* Sparse allreduce collective related fields */
    ccl_sparse_allreduce_completion_fn_t sparse_allreduce_completion_fn;
    /* User context for saving sparse_allreduce results */
    const void* sparse_allreduce_completion_ctx;
    /* Priority for collective operation */
    size_t priority;
    /* Blocking/non-blocking */
    int synchronous;
    /* Persistent/non-persistent */
    int to_cache;
    /* Treat buffer as vector/regular - applicable for allgatherv only */
    int vector_buf;
    /**
    * Id of the operation. If specified, new communicator is created and colle
    * operations with the same @b match_id are executed in the same o
    */
    const char* match_id;
} ccl_coll_attr_t;
```

`ccl_coll_attr_t` (`ccl::coll_attr` in C++ version of API) is an extendable structure that serves as a modificator of communication primitive behaviour. It can be optionally passed into any collective operation exposed by oneCCL.

Track Communication Progress

You can track the progress for any of the collective operations provided by oneCCL using the Test or Wait function for the Request object.

Request

Each collective communication operation of oneCCL returns a request that can be used to query completion of this operation or to block the execution while the operation is in progress. oneCCL request is an opaque handle that is managed by corresponding APIs.

C API C++ API

```
/**
* A request interface that allows the user to track collective operation progress
*/
class request
{
public:
    /**
    * Blocking wait for collective operation completion
    */
    virtual void wait() = 0;

    /**
    * Non-blocking check for collective operation completion
    * @retval true if the operations has been completed
    * @retval false if the operations has not been completed
    */
    virtual bool test() = 0;

    virtual ~request() = default;
};
```

Test

Non-blocking operation that returns the completion status.

C API C++ API

```
bool request::test();
```

Returnes the value that indicates the status:

- 0 - operation is in progress
- otherwise, the operation is completed

Wait

Operation that blocks the execution until communication operation is completed.

C API C++ API

```
void request::wait();
```

Error Handling

Error handling in oneCCL is implemented differently in C and C++ versions of API. C version of API uses error codes that are returned by every exposed function, while C++ API uses exceptions.

C API C++ API

```
class ccl_error : public std::runtime_error
```

Generic Workflow

Below is a generic flow for using C++ API of oneCCL:

1. Initialize the library:

   ```
   ccl::environment::instance();
   ```

   Alternatively, you can create communicator objects:

   ```
   ccl::communicator_t comm = ccl::environment::instance().create_communicator();
   ```

2. Execute collective operation of choice on this communicator:

   ```
   auto request = comm.allreduce(...);
   request->wait();
   ```

GPU support

You can choose between CPU and GPU backends by specifying `ccl_stream_type` value during the ccl stream object creation:

- For GPU backend, specify `ccl_stream_sycl` as the first argument.
- For collective operations, which operate on SYCL* stream, C version of oneCCL API expects communication buffers to be `sycl::buffer*` objects casted to `void*`.

The example below demonstrates these concepts.

Example

Consider a simple `allreduce` example for GPU.

1. Create a GPU ccl stream object:

   C API | **C++ API**

   ```
   ccl::stream_t stream =
   ccl::environment::instance().create_stream(cc::stream_type::sycl, &q);
   ```

   `q` is an object of type `sycl::queue`.

2. To illustrate the `ccl_allreduce` execution, initialize `sendbuf` (in real scenario it is provided by application):

   ```
   auto host_acc_sbuf = sendbuf.get_access<mode::write>();
   for (i = 0; i < COUNT; i++) {
       host_acc_sbuf[i] = rank;
   }
   ```

3. For demostration purposes only, modify the `sendbuf` on the GPU side:

```
q.submit([&](cl::sycl::handler& cgh) {
    auto dev_acc_sbuf = sendbuf.get_access<mode::write>(cgh);
    cgh.parallel_for<class allreduce_test_sbuf_modify>(range<1>{COUNT}, [=](item<1>
id) {
        dev_acc_sbuf[id] += 1;
    });
});
```

`ccl_allreduce` invocation performs reduction of values from all processes and then distributes the result to all processes. In this case, the result is an array with the size equal to the number of processes ($\#processes$), where all elements are equal to the sum of arithmetical progression:

$$\#processes \cdot (\#processes - 1)/2$$

C API

C++ API

```
comm.allreduce(sendbuf,
               recvbuf,
               COUNT,
               ccl::reduction::sum,
               nullptr, /* attr */
               stream)->wait();
```

4. Check the correctness of `ccl_allreduce` on the GPU:

```
q.submit([&](handler& cgh) {
    auto dev_acc_rbuf = recvbuf.get_access<mode::write>(cgh);
    cgh.parallel_for<class allreduce_test_rbuf_check>(range<1>{COUNT}, [=](item<1>
id) {
        if (dev_acc_rbuf[id] != size*(size+1)/2) {
        dev_acc_rbuf[id] = -1;
        }
    });
});
```

```
if (rank == COLL_ROOT) {
    auto host_acc_rbuf_new = recvbuf.get_access<mode::read>();
    for (i = 0; i < COUNT; i++) {
        if (host_acc_rbuf_new[i] == -1) {
            cout << "FAILED" << endl;
            break;
        }
    }
    if (i == COUNT) {
        cout<<"PASSED"<<endl;
    }
}
```

Note

When using C version of oneCCL API, it is required to explicitly free the created GPU ccl stream object:

```
ccl_stream_free(stream);
```

For C++ version of oneCCL API this is performed implicitly.

CPU support

You can choose between CPU and GPU backends by specifying the `ccl_stream_type` value during ccl stream object creation.

- For CPU backend, specify the `ccl_stream_cpu` value.
- For collective operations performed using CPU stream, oneCCL expects communication buffers to reside in the host memory.

The example below demonstrates these concepts.

Example

Consider a simple `allreduce` example for CPU.

1. Create a CPU ccl stream object:

   C API   **C++ API**

   ```
   /* For CPU, NULL is passed instead of native stream pointer */
   ccl::stream_t stream =
   ccl::environment::instance().create_stream(cc::stream_type::cpu, NULL);
   ```

   or just

   ```
   ccl::stream stream;
   ```

2. To illustrate the `ccl_allreduce` execution, initialize `sendbuf` (in real scenario it is supplied by application):

```
/* initialize sendbuf */
for (i = 0; i < COUNT; i++) {
    sendbuf[i] = rank;
}
```

`ccl_allreduce` invocation performs reduction of values from all processes and then distributes the result to all processes. In this case, the result is an array with the size equal to the number of processes (#processes), where all elements are equal to the sum of arithmetical progression:

$$\#\text{processes} \cdot (\#\text{processes} - 1)/2$$

C API C++ API

```
comm.allreduce(&sendbuf,
               &recvbuf,
               COUNT,
               ccl::reduction::sum,
               nullptr, /* attr */
               stream)->wait();
```

Note

When using C version of oneCCL API, it is required to explicitly free ccl stream object:

```
ccl_stream_free(stream);
```

For C++ version of oneCCL API this is performed implicitly.

Execution of collective operations

Collective operations are executed by CCL worker threads (workers). The number of workers is controlled by the CCL_WORKER_COUNT environment variable.

Workers affinity is controlled by CCL_WORKER_AFFINITY.

By setting workers affinity you can specify which CPU cores are used to host CCL workers. The general rule of thumb is to use different CPU cores for compute (e.g. by specifying `KMP_AFFINITY`) and for communication.

There are two ways to set workers affinity: explicit and automatic.

Explicit setup

To set affinity explicitly, pass ID of the cores to be bound to to the `CCL_WORKER_AFFINITY` environment variable.

Example

In the example below, oneCCL creates 4 threads and pins them to cores with numbers 3, 4, 5, and 6, respectively:

```
export CCL_WORKER_COUNT=4
export CCL_WORKER_AFFINITY=3,4,5,6
```

Automatic setup

> Note
>
> Automatic pinning only works if application is launched using `mpirun` provided by the oneCCL distribution package.

To set affinity automatically, set `CCL_WORKER_AFFINITY` to `auto`.

Example

In the example below, oneCCL creates four threads and pins them to the last four cores available for the process launched:

```
export CCL_WORKER_COUNT=4
export CCL_WORKER_AFFINITY=auto
```

> **Note**
>
> The exact IDs of CPU cores depend on the parameters passed to `mpirun`.

Transport selection

oneCCL supports two transports for inter-node communication: Intel® MPI Library and libfabrics.

The transport selection is controlled by CCL_ATL_TRANSPORT.

In case of MPI over libfaric implementation (for example, Intel® MPI Library 2019) or in case of direct libfabric transport, the selection of specific libfaric provider is controlled by the `FI_PROVIDER` environment variable.

Selection of collective algorithms

oneCCL supports manual selection of collective algorithms for different message size ranges. Please refer to the Collective algorithms selection section for details.

Caching of collective operations

Collective operations may have expensive initialization phase (for example, allocation of internal structures and buffers, registration of memory buffers, handshake with peers, and so on). oneCCL amortizes these overheads by caching collective internal representations and reusing them on the subsequent calls.

To control this, set `coll_attr.to_cache = 1` and `coll_attr.match_id = <match_id>`, where `<match_id>` is a unique string (for example, tensor name). Note that:

- `<match_id>` should be the same for a specific collective operation across all ranks.
- If the same tensor is a part of different collective operations, `match_id` should have different values for each of these operations.

Fusion of collective operations

In some cases, it may be beneficial to postpone execution of collective operations and execute them all together as a single operation in a batch mode. This can reduce operation setup overhead and improve interconnect saturation.

oneCCL provides several knobs to enable and control such optimization:

- The fusion is enabled by CCL_FUSION.
- The advanced configuration is controlled by:
  - CCL_FUSION_BYTES_THRESHOLD
  - CCL_FUSION_COUNT_THRESHOLD
  - CCL_FUSION_CYCLE_MS

For now, this functionality is supported for allreduce operations only.

Sparse collective operations

Language models typically feature huge embedding tables within their topology. This makes straight-forward gradient computation followed by `allreduce` for the whole set of weights not feasible in practice due to both performance and memory footprint reasons. Thus, gradients for such layers are usually computed for a smaller sub-tensor on each iteration, and communication pattern, which is required to average the gradients across processes, does not map well to allreduce API.

To address these scenarios, frameworks usually utilize the `allgather` primitive, which may be suboptimal if there is a lot of intersections between sub-tensors from different processes.

Latest research paves the way to handling such communication in a more optimal manner, but each of these approaches has its own application area. The ultimate goal of oneCCL is to provide a common API for sparse collective operations that would simplify framework design by allowing under-the-hood implementation of different approaches.

oneCCL can work with sparse tensors represented by two tensors: one for indices and one for values.

The `sparse_allreduce` function has the following parameters:

- `send_ind_buf` - a buffer of indices with `send_ind_count` elements of `index_dtype`
- `send_int_count` - the number of `send_ind_buf` elements of type `index_type`
- `send_val_buf` - a buffer of values with `send_val_count` elements of `value_dtype`
- `send_val_count` - the number of `send_val_buf` elements of type `value_type`
- `recv_ind_buf` - a buffer to store reduced indices (ignored for now)
- `recv_ind_count` - the number of reduced indices (ignored for now)
- `` recv_val_buf` `` - a buffer to store reduced values (ignored for now)
- `recv_val_count` - the number of reduced values (ignored for now)
- `index_dtype` - index type of elements in `send_ind_buf` and `recv_ind_buf` buffers
- `value_dtype` - data type of elements in `send_val_buf` and `recv_val_buf` buffers
- `reduction` - the type of reduction operation to be applied
- `attributes` - attributes that customize operation
- returns `ccl::request` object to track the progress of the operation

For `sparse_allreduce`, a completion callback is required to get the results. Use the following Collective Call Attributes fields:

- `sparse_allreduce_completion_fn` - a completion callback function pointer (must not be set to `NULL`)
- `sparse_allreduce_completion_ctx` - a user context pointer of type `void*`

Here is an example of a function definition for `sparse_allreduce` completion callback:

```
ccl_status_t sparse_allreduce_completion_fn(
    const void* indices_buf, size_t indices_count, ccl_datatype_t indices_datatype,
    const void* values_buf, size_t values_count, ccl_datatype_t values_datatype,
    const ccl_fn_context_t* fn_ctx, const void* user_ctx)
{
    /*
      Note that indices_buf and values_buf are temporary buffers.
      Thus, the data from these buffers should be copied. Use user_ctx for
      this purpose.
    */
    return ccl_status_success;
}
```

For more details, refer to this example

Unordered collectives support

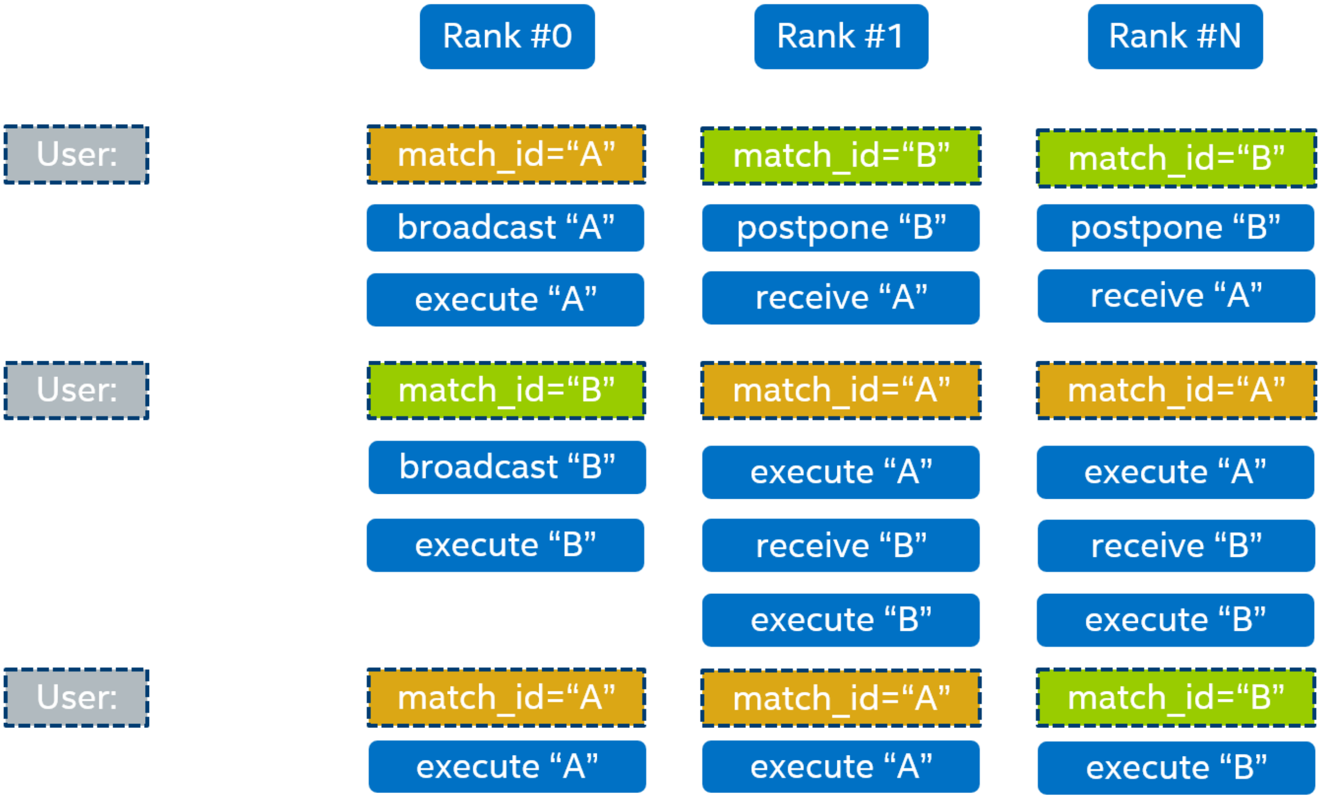
Some deep learning frameworks deploy local scheduling approach for the graph of operations, which may result in different ordering of collective operations across different processes. When using communication middleware that requires the same order of collective calls across different ranks, such scenarios may result in hangs or data corruption. This requires complicated coordination logic to maintain the same ordering.

In contrast, oneCCL provides a mechanism to arrange execution of collective operations in accordance with the user-defined identifier.

To set an identifier, use `ccl::coll_attr.match_id`, where `match_id` is a pointer to a null-terminated C-style string.

Unordered collectives' execution is coordinated by the zero-id rank (root rank). When root rank receives a user request with a non-empty `match_id` for the first time, it broadcasts information about the user identifier to all other ranks and assigns an internal oneCCL identifier that will later be used with all following operations with the same `match_id`.

When a non-root rank receives a user request with a non-empty `match_id` for the first time, it postpones operation execution until it receives a message from the root rank. Once the message is received, the rank creates an internal oneCCL identifier that will be used for all following operations with the same `match_id`.



	Rank #0	Rank #1	Rank #N
User:	match_id=“A”	match_id=“B”	match_id=“B”
	broadcast “A”	postpone “B”	postpone “B”
	execute “A”	receive “A”	receive “A”
User:	match_id=“B”	match_id=“A”	match_id=“A”
	broadcast “B”	execute “A”	execute “A”
	execute “B”	receive “B”	receive “B”
		execute “B”	execute “B”
User:	match_id=“A”	match_id=“A”	match_id=“B”
	execute “A”	execute “A”	execute “B”

Fault tolerance / elasticity

Main instructions

Start with setting CCL_ATL_TRANSPORT to *ofi*.

Before launching ranks, you can specify CCL_WORLD_SIZE = N, where N is the number of ranks to start. If k8s with k8s manager support is used, then N is equal to `replicasize` by default.

You can specify your own function that decides what oneCCL should do on the “world” resize event:

- wait
- use current “world” information
- finalize

```
typedef enum ccl_resize_action
{
  // wait additional changes for number of ranks
  ccl_ra_wait     = 0,

  // run with current number of ranks
  ccl_ra_use      = 1,

  // finalize work
  ccl_ra_finalize = 2,

} ccl_resize_action_t;

typedef ccl_resize_action_t(*ccl_resize_fn_t)(size_t comm_size);

set_resize_fn(ccl_resize_fn_t callback);
```

In case the number of ranks is changed, this function is called on oneCCL level. Application level (e.g. framework) should return the action that oneCCL should perform.

Setting this function to `NULL` (default value) means that oneCCL will work with exactly CCL_WORLD_SIZE or `replicasize` ranks without fault tolerant / elasticity.

Examples

Without k8s manager

To run ranks in k8s without k8s manager, for example, set of pods:

- CCL_PM_TYPE = resizable
- CCL_K8S_API_ADDR = k8s server address and port (in a format of IP:PORT)
- Set same label CCL_JOB_NAME = job_name on each pod
- Run your example

Using k8s manager

To run ranks in k8s use statefulset / deployment as a manager:

- CCL_PM_TYPE = resizable
- CCL_K8S_API_ADDR = k8s server address
- CCL_K8S_MANAGER_TYPE = k8s
- Run your example

Without mpirun

To run ranks without `mpirun`:

- CCL_PM_TYPE = resizable
- CCL_KVS_IP_EXCHANGE = env
- CCL_KVS_IP_PORT = ip_port of one of your nodes where you run the example
- Run your example

Environment variables

Collective algorithms selection

CCL_<coll_name>

Syntax

To set a specific algorithm for the whole message size range:

```
CCL_<coll_name>=<algo_name>
```

To set a specific algorithm for a specific message size range:

```
CCL_<coll_name>="<algo_name_1>[:<size_range_1>][;<algo_name_2>:<size_range_2>][;...]"
```

Where:

- `<coll_name>` is selected from a list of available collective operations (Available collectives).
- `<algo_name>` is selected from a list of available algorithms for a specific collective operation (Available algorithms).
- `<size_range>` is described by the left and the right size borders in a format `<left>-<right>`. Size is specified in bytes. Use reserved word `max` to specify the maximum message size.

oneCCL internally fills algorithm selection table with sensible defaults. User input complements the selection table. To see the actual table values set `CCL_LOG_LEVEL=1`.

Example

```
CCL_ALLREDUCE="recursive_doubling:0-8192;rabenseifner:8193-1048576;ring:1048577-max"
```

Available collectives

Available collective operations (`<coll_name>`):

- `ALLGATHERV`
- `ALLREDUCE`
- `ALLTOALL`
- `ALLTOALLV`
- `BARRIER`
- `BCAST`
- `REDUCE`
- `SPARSE_ALLREDUCE`

Available algorithms

Available algorithms for each collective operation (`<algo_name>`):

`ALLGATHERV` algorithms

`direct`	Based on `MPI_Iallgatherv`
`naive`	Send to all, receive from all
`flat`	Alltoall-based algorithm
`multi_bcast`	Series of broadcast operations with different root ranks

`ALLREDUCE` algorithms

`direct`	Based on `MPI_Iallreduce`
`rabenseifner`	Rabenseifner's algorithm
`starlike`	May be beneficial for imbalanced workloads
`ring`	reduce_scatter+allgather ring. Use `CCL_RS_CHUNK_COUNT` and `CCL_RS_M`
`ring_rma`	reduce_scatter+allgather ring using RMA communications
`double_tree`	Double-tree algorithm
`recursive_doubling`	Recursive doubling algorithm
`2d`	2-dimensional algorithm (reduce_scatter+allreduce+allgather)

`ALLTOALL` algorithms

`direct`	Based on `MPI_Ialltoall`
`naive`	Send to all, receive from all

`ALLTOALLV` algorithms

`direct`	Based on `MPI_Ialltoallv`
`naive`	Send to all, receive from all

`BARRIER` algorithms

`direct`	Based on `MPI_Ibarrier`
`ring`	Ring-based algorithm

`BCAST` algorithms

`direct`	Based on `MPI_Ibcast`
`ring`	Ring
`double_tree`	Double-tree algorithm
`naive`	Send to all from root rank

`REDUCE` algorithms

`direct`	Based on `MPI_Ireduce`
`rabenseifner`	Rabenseifner’s algorithm
`tree`	Tree algorithm
`double_tree`	Double-tree algorithm

`SPARSE_ALLREDUCE` algorithms

`ring`	Ring-allreduce based algorithm
`mask`	Mask matrix based algorithm
`allgatherv`	3-allgatherv based algorithm

CCL_RS_CHUNK_COUNT

Syntax

```
CCL_RS_CHUNK_COUNT=<value>
```

Arguments

<value>	Description
`COUNT`	Maximum number of chunks.

Description

Set this environment variable to specify maximum number of chunks for reduce_scatter phase in ring allreduce.

CCL_RS_MIN_CHUNK_SIZE

Syntax

```
CCL_RS_MIN_CHUNK_SIZE=<value>
```

Arguments

<value>	Description
`SIZE`	Minimum number of bytes in chunk.

Description

Set this environment variable to specify minimum number of bytes in chunk for reduce_scatter phase in ring allreduce. Affects actual value of `CCL_RS_CHUNK_COUNT`.

Fusion

CCL_FUSION

Syntax

```
CCL_FUSION=<value>
```

Arguments

<value>	Description
`1`	Enable fusion of collective operations
`0`	Disable fusion of collective operations (**default**)

Description

Set this environment variable to control fusion of collective operations. The real fusion depends on additional settings described below.

CCL_FUSION_BYTES_THRESHOLD

Syntax

```
CCL_FUSION_BYTES_THRESHOLD=<value>
```

Arguments

<value>	Description
`SIZE`	Bytes threshold for a collective operation. If the size of a communication buffer in

Description

Set this environment variable to specify the threshold of the number of bytes for a collective operation to be fused.

CCL_FUSION_COUNT_THRESHOLD

Syntax

```
CCL_FUSION_COUNT_THRESHOLD=<value>
```

Arguments

<value>	Description
`COUNT`	The threshold for the number of collective operations. oneCCL can fuse together

Description

Set this environment variable to specify count threshold for a collective operation to be fused.

CCL_FUSION_CYCLE_MS

Syntax

```
CCL_FUSION_CYCLE_MS=<value>
```

Arguments

<value>	Description
`MS`	The frequency of checking for collectives operations to be fused, in milliseconds: • Small `MS` value can improve latency. • Large `MS` value can help to fuse larger number of operations at a time.

Description

Set this environment variable to specify the frequency of checking for collectives operations to be fused.

PMI

CCL_PM_TYPE

Syntax

```
CCL_PM_TYPE=<value>
```

Arguments

<value>	Description
`simple`	Use PMI (process manager interface) with `mpirun` (**default**).
`resizable`	Use internal KVS (key-value storage) without `mpirun`.

Description

Set this environment variable to specify the process manager type.

CCL_KVS_IP_EXCHANGE

Syntax

```
CCL_KVS_IP_EXCHANGE=<value>
```

Arguments

<value>	Description
`k8s`	Use K8S for IP exchange (**default**).

<value>	Description
`env`	Use a specific environment to get the master IP.

Description

Set this environment variable to specify the way to IP addresses of ran processes are exchanged.

CCL_K8S_API_ADDR

Syntax

```
CCL_K8S_API_ADDR =<value>
```

Arguments

<value>	Description
`IP:PORT`	Set the address and the port of k8s kvs.

Description

Set this environment variable to specify k8s kvs address.

CCL_K8S_MANAGER_TYPE

Syntax

```
CCL_K8S_MANAGER_TYPE=<value>
```

Arguments

<value>	Description
`none`	Use Pods labels for IP exchange (**default**).

<value>	Description
k8s	Use StatefulsetDeployment labels for IP exchange.

Description

Set this environment variable to specify the way of IP exchange.

CCL_KVS_IP_PORT

Syntax

```
CCL_KVS_IP_PORT=<value>
```

Arguments

<value>	Description
IP_PORT	Set the address and the port of the master kvs server.

Description

Set this environment variable to specify the master kvs address.

CCL_WORLD_SIZE

Syntax

```
CCL_WORLD_SIZE=<value>
```

Arguments

<value>	Description
N	The number of processes to start execution.

Description

Set this environment variable to specify the number of oneCCL processes.

CCL_JOB_NAME

Syntax

```
CCL_JOB_NAME=<value>
```

Arguments

<value>	Description
`job_name`	The name of the job.

Description

Set this label on the pods that should be connected with each other.

CCL_ATL_TRANSPORT

Syntax

```
CCL_ATL_TRANSPORT=<value>
```

Arguments

<value>	Description
`mpi`	MPI transport (**default**).
`ofi`	OFI (libfaric) transport.

Description

Set this environment variable to select the transport for inter-node communications.

CCL_UNORDERED_COLL

Syntax

```
CCL_UNORDERED_COLL=<value>
```

Arguments

<value>	Description
`1`	Enable execution of unordered collectives. You have to additionally specify `coll_`
`0`	Disable execution of unordered collectives (**default**).

Description

Set this environment variable to enable execution of unordered collective operations on different nodes.

CCL_PRIORITY

Syntax

```
CCL_PRIORITY=<value>
```

Arguments

<value>	Description
`direct`	You have to explicitly specify priority using `coll_attr.priority`.
`lifo`	Priority is implicitly increased on each collective call. You do not have to specify p
`none`	Disable prioritization (**default**).

Description

Set this environment variable to control priority mode of collective operations.

CCL_WORKER_COUNT

Syntax

```
CCL_WORKER_COUNT=<value>
```

Arguments

<value>	Description
`N`	The number of worker threads for oneCCL rank (`1` if not specified).

Description

Set this environment variable to specify the number of oneCCL worker threads.

CCL_WORKER_AFFINITY

Syntax

```
CCL_WORKER_AFFINITY=<proclist>
```

Arguments

<proclist>	Description
`n1,n2,..`	Affinity is explicitly specified by a user.
`auto`	Workers are pinned to K last cores of pin domain, where K is `CCL_WORKER_COUNT`

Description

Library API

Class Hierarchy

- **Namespace ccl**
  - Template Struct api_type_info
  - Template Struct ccl_host_attributes_traits
  - Template Struct ccl_host_attributes_traits< ccl_host_color >
  - Template Struct ccl_host_attributes_traits< ccl_host_version >
  - Class ccl_error
  - Class ccl_host_attr
  - Class communicator
  - Class environment
  - Class request
  - Class stream
  - Enum datatype
  - Enum reduction
  - Enum stream_type
- Struct ccl_coll_attr_t
- Struct ccl_comm_attr_t
- Struct ccl_comm_attr_versioned_t
- Struct ccl_datatype_attr_t
- Struct ccl_fn_context_t
- Struct ccl_version_t
- Enum ccl_host_attributes
- Enum ccl_reduction_t
- Enum ccl_resize_action
- Enum ccl_status_t
- Enum ccl_stream_type_t

File Hierarchy

- **Directory include**
    - File ccl.h
    - File ccl.hpp
    - File ccl_device_type_traits.hpp
    - File ccl_device_types.h
    - File ccl_device_types.hpp
    - File ccl_gpu_modules.h
    - File ccl_type_traits.hpp
    - File ccl_types.h
    - File ccl_types.hpp
    - File gpu_communicator.hpp

Full API

Namespaces

- Namespace ccl
    - Classes
    - Enums
    - Functions
    - Typedefs

Classes and Structs

- Template Struct api_type_info
    - Struct Documentation

- Template Struct ccl_host_attributes_traits
    - Struct Documentation

- Template Struct ccl_host_attributes_traits< ccl_host_color >
    - Struct Documentation

- Template Struct ccl_host_attributes_traits< ccl_host_version >
    - Struct Documentation

Enums

Functions

Defines

Typedefs

Docs » Library API » Namespace ccl

Namespace ccl

Contents

Classes

- Template Struct api_type_info
- Template Struct ccl_host_attributes_traits
- Template Struct ccl_host_attributes_traits< ccl_host_color >
- Template Struct ccl_host_attributes_traits< ccl_host_version >
- Class ccl_error
- Class ccl_host_attr
- Class communicator
- Class environment
- Class request
- Class stream

Enums

- Enum datatype
- Enum reduction
- Enum stream_type

Functions

- Function ccl::datatype_create
- Function ccl::datatype_free
- Function ccl::datatype_get_size
- Template Function ccl::is_attribute_value_supported
- Template Function ccl::is_class
- Template Function ccl::is_class_supported
- Template Function ccl::is_native_type_supported

- Template Function ccl::is_stream_supported
- Template Function ccl::is_supported

Typedefs

- Typedef ccl::bfp16
- Typedef ccl::coll_attr
- Typedef ccl::comm_attr
- Typedef ccl::comm_attr_t
- Typedef ccl::communicator_t
- Typedef ccl::datatype_attr
- Typedef ccl::shared_communicator_t
- Typedef ccl::stream_t

Template Struct api_type_info

- Defined in File ccl_device_type_traits.hpp

Struct Documentation

```
template<class type>
struct api_type_info
```

Public Static Functions

> *static constexpr* bool **is_supported**()

> *static constexpr* bool **is_class**()

Template Struct ccl_host_attributes_traits

- Defined in File ccl_types.hpp

Struct Documentation

```
template<ccl_host_attributesattrId>
struct ccl_host_attributes_traits
```

Template Struct ccl_host_attributes_traits< ccl_host_color >

- Defined in File ccl_type_traits.hpp

Struct Documentation

template<>
struct **ccl_host_attributes_traits**<ccl_host_color>

Enumeration of supported CCL API data types

Public Types

> template<>
> *using* **type** = int

Template Struct ccl_host_attributes_traits< ccl_host_version >

- Defined in File ccl_type_traits.hpp

Struct Documentation

template<>
struct **ccl_host_attributes_traits**<ccl_host_version>

Public Types

> template<>
> *using* **type** = ccl_version_t

Template Struct ccl_host_attributes_traits< ccl_host_version >

- Defined in File ccl_type_traits.hpp

Struct Documentation

template<>
struct `ccl_host_attributes_traits`<ccl_host_version>

Public Types

> template<>
> *using* `type` = ccl_version_t

Struct ccl_coll_attr_t

- Defined in File ccl_types.h

Struct Documentation

struct **ccl_coll_attr_t**

Extendable list of collective attributes.

Public Members

ccl_prologue_fn_t**prologue_fn**

Callbacks into application code for pre-/post-processing data and custom reduction operation

ccl_epilogue_fn_t**epilogue_fn**

ccl_reduction_fn_t**reduction_fn**

ccl_sparse_allreduce_completion_fn_t**sparse_allreduce_completion_fn**

const void ***sparse_allreduce_completion_ctx**

size_t **priority**

int **synchronous**

int **to_cache**

int **vector_buf**

const char ***match_id**

Id of the operation. If specified, new communicator will be created and collective operations with the same **match_id** will be executed in the same order.

Struct ccl_comm_attr_t

- Defined in File ccl_types.h

Struct Documentation

struct **ccl_comm_attr_t**

List of host communicator attributes.

Public Members

int **color**

Used to split global communicator into parts. Ranks with identical color will form a new communicator.

size_t ***ranks**

size_t **size**

const size_t ***dev_list**

int **local**

Struct ccl_comm_attr_versioned_t

- Defined in File ccl_types.h

Struct Documentation

struct **ccl_comm_attr_versioned_t**

Public Members

ccl_comm_attr_t**comm_attr**

int **version**

Struct ccl_datatype_attr_t

- Defined in File ccl_types.h

Struct Documentation

struct **ccl_datatype_attr_t**

Public Members

size_t **size**

Struct ccl_fn_context_t

- Defined in File ccl_types.h

Struct Documentation

struct **ccl_fn_context_t**

Public Members

const char ***match_id**

const size_t **offset**

Struct ccl_version_t

- Defined in File ccl_types.h

Struct Documentation

struct `ccl_version_t`

API version description.

Public Members

unsigned int `major`

unsigned int `minor`

unsigned int `update`

const char *`product_status`

const char *`build_date`

const char *`full`

Class ccl_error

- Defined in File ccl_types.hpp

Inheritance Relationships

Base Type

- `public runtime_error`

Class Documentation

class `ccl_error` : *public* runtime_error

Exception type that may be thrown by ccl API

Public Functions

`ccl_error`(*const* std::string &*message*)

`ccl_error`(*const* char **message*)

Class ccl_host_attr

- Defined in File ccl.hpp

Class Documentation

class **ccl_host_attr**

Class `ccl_host_attr` allows to configure host wire-up communicator creation parametes

Public Functions

virtual **~ccl_host_attr**()

template<ccl_host_attributes**attrId**, class **Value**, class = *typename* std::enable_if<is_attribute_value_supported<attrId, Value>()>::type>
Value**set_value**(*const* Value &*v*)

Set specific value for attribute by @attrId. Previous attibute value would be returned

template<ccl_host_attributes**attrId**>
const ccl_host_attributes_traits<attrId>::type &**get_value**()*const*

Get specific attribute value by @attrId

Protected Functions

ccl_host_attr(*const* ccl_host_attr &*src*)

Friends

friend **ccl::ccl_host_attr::ccl_device_attr**

friend **ccl::ccl_host_attr::communicator_interface_dispatcher**

friend **ccl::ccl_host_attr::environment**

Class communicator

- Defined in File ccl.hpp

Class Documentation

class **`communicator`**

A communicator that permits collective operations Has no defined public constructor. Use ccl::environment::create_communicator or ccl::comm_group for communicator objects creation

Public Types

using **`coll_request_t`** = std::unique_ptr<request>

Type allows to operate request interface in RAII manner

Public Functions

`~communicator`()

size_t **`rank`**()*const*

Retrieves the rank of the current process in a communicator

Return

rank of the current process

size_t **`size`**()*const*

Retrieves the number of processes in a communicator

Return

number of the processes

comm_attr_t**`get_host_attr`**()*const*

bool **`is_ready`**()*const*

Retrieves status of wiring-up progress of communicators in group After all expected communicators are created in parent comm_group, then wiring-up phase is automatically executed and all communicator object will go in ready status

coll_request_t**allgatherv**(*const* void **send_buf*, size_t *send_count*, void **recv_buf*, *const* size_t **recv_counts*, ccl::datatype*dtype*, *const* ccl::coll_attr **attr* = nullptr, *const* ccl::stream_t &*stream* = ccl::stream_t())

Gathers `buf` on all process in the communicator and stores result in `recv_buf` on each process

Return

ccl::communicator::coll_request_t object that can be used to track the progress of the operation

Parameters

- `send_buf`: the buffer with `count` elements of `dtype` that stores local data to be gathered
- `send_count`: number of elements of type `dtype` in `send_buf`
- `recv_buf`: [out] the buffer to store gathered result on the `each` process, must have the same dimension as `buf`. Used by the `root` process only, ignored by other processes
- `recv_counts`: array with number of elements received by each process
- `dtype`: data type of elements in the buffer `buf` and `recv_buf`
- `attr`: optional attributes that customize operation

template<class **buffer_type**, class = *typename* std::enable_if<ccl::is_native_type_supported<buffer_type>()>::type> coll_request_t**allgatherv**(*const* buffer_type **send_buf*, size_t *send_count*, buffer_type **recv_buf*, *const* size_t **recv_counts*, *const* ccl::coll_attr **attr* = nullptr, *const* ccl::stream_t &*stream* = ccl::stream_t())

Type safety version: Gathers `buf` on all process in the communicator and stores result in `recv_buf` on each process

Return

ccl::communicator::coll_request_t object that can be used to track the progress of the operation

Parameters

- `send_buf`: the buffer with `count` elements of `buffer_type` that stores local data to be gathered

- `send_count`: number of elements of type `buffer_type` in `send_buf`
- `recv_buf`: [out] the buffer to store gathered result on the `each` process, must have the same dimension as `buf`. Used by the `root` process only, ignored by other processes
- `recv_counts`: array with number of elements received by each process
- `attr`: optional attributes that customize operation

> template<class **buffer_container_type**, class = *typename* std::enable_if<ccl::is_class_supported<buffer_container_type>()>::type>
> coll_request_t**allgatherv**(*const* buffer_container_type &*send_buf*, size_t *send_count*, buffer_container_type &*recv_buf*, *const* size_t **recv_counts*, *const* ccl::coll_attr **attr* = nullptr, *const* ccl::stream_t &*stream* = ccl::stream_t())

Type safety version: Gathers `buf` on all process in the communicator and stores result in `recv_buf` on each process

Return

ccl::request object that can be used to track the progress of the operation

Parameters

- `send_buf`: the buffer of `buffer_container_type` with `count` elements that stores local data to be gathered
- `send_count`: number of elements in `send_buf`
- `recv_buf`: [out] the buffer of `buffer_container_type` to store gathered result on the `each` process, must have the same dimension as `buf`. Used by the `root` process only, ignored by other processes
- `recv_counts`: array with number of elements received by each process
- `attr`: optional attributes that customize operation

> coll_request_t**allreduce**(*const* void **send_buf*, void **recv_buf*, size_t *count*, ccl::datatype*dtype*, ccl::reduction*reduction*, *const* ccl::coll_attr **attr* = nullptr, *const* ccl::stream_t &*stream* = ccl::stream_t())

Reduces `buf` on all process in the communicator and stores result in `recv_buf` on each process

Return

ccl::communicator::coll_request_t object that can be used to track the progress of the operation

Parameters

- `send_buf` : the buffer with `count` elements of `dtype` that stores local data to be reduced
- `recv_buf` : [out] - the buffer to store reduced result , must have the same dimension as `buf` .
- `count` : number of elements of type `dtype` in `buf`
- `dtype` : data type of elements in the buffer `buf` and `recv_buf`
- `reduction` : type of reduction operation to be applied
- `attr` : optional attributes that customize operation

template<class **buffer_type**, class = *typename* std::enable_if<ccl::is_native_type_supported<buffer_type>()>::type>
coll_request_t**allreduce**(*const* buffer_type **send_buf*, buffer_type **recv_buf*, size_t *count*, ccl::reduction*reduction*, *const* ccl::coll_attr **attr* = nullptr, *const* ccl::stream_t &*stream* = ccl::stream_t())

Type safety version: Reduces `buf` on all process in the communicator and stores result in `recv_buf` on each process

Return

ccl::communicator::coll_request_t object that can be used to track the progress of the operation

Parameters

- `send_buf` : the buffer with `count` elements of `buffer_type` that stores local data to be reduced
- `recv_buf` : [out] - the buffer to store reduced result , must have the same dimension as `buf` .
- `count` : number of elements of type `buffer_type` in `buf`
- `reduction` : type of reduction operation to be applied
- `attr` : optional attributes that customize operation

template<class **buffer_container_type**, class = *typename* std::enable_if<ccl::is_class_supported<buffer_container_type>()>::type>
coll_request_t**allreduce**(*const* buffer_container_type &*send_buf*, buffer_container_type &*recv_buf*, size_t *count*, ccl::reduction*reduction*, *const* ccl::coll_attr **attr* = nullptr, *const* ccl::stream_t &*stream* = ccl::stream_t())

Type safety version: Reduces `buf` on all process in the communicator and stores result in `recv_buf` on each process

Return

ccl::communicator::coll_request_t object that can be used to track the progress of the operation

Parameters

- `send_buf`: the buffer of `buffer_container_type` with `count` elements that stores local data to be reduced
- `recv_buf`: [out] - the buffer of `buffer_container_type` to store reduced result, must have the same dimension as `buf`.
- `count`: number of elements in `send_buf`
- `reduction`: type of reduction operation to be applied
- `attr`: optional attributes that customize operation

coll_request_t**alltoall**(*const* void **send_buf*, void **recv_buf*, size_t *count*, ccl::datatype*dtype*, *const* ccl::coll_attr **attr* = nullptr, *const* ccl::stream_t &*stream* = ccl::stream_t())

Each process sends distinct data to each of the receivers. The j-th block sent from process i is received by process j and is placed in the i-th block of recvbuf.

Return

ccl::request object that can be used to track the progress of the operation

Parameters

- `send_buf`: the buffer with `count` elements of `dtype` that stores local data
- `recv_buf`: [out] - the buffer to store received result , must have the N * dimension of `buf`, where N - communicator size.
- `count`: number of elements to send / receive from each process
- `dtype`: data type of elements in the buffer `buf` and `recv_buf`
- `attr`: optional attributes that customize operation

template<class **buffer_type**, class = *typename* std::enable_if<ccl::is_native_type_supported<buffer_type>()>::type> coll_request_t**alltoall**(*const* buffer_type **send_buf*, buffer_type **recv_buf*, size_t *count*, *const* ccl::coll_attr **attr* = nullptr, *const* ccl::stream_t &*stream* = ccl::stream_t())

Each process sends distinct data to each of the receivers. The j-th block sent from process i is received by process j and is placed in the i-th block of recvbuf.

Return

ccl::request object that can be used to track the progress of the operation

> Parameters

- `send_buf`: the buffer with `count` elements of `dtype` that stores local data
- `recv_buf`: [out] - the buffer to store received result , must have the N * dimension of `buf`, where N - communicator size.
- `count`: number of elements to send / receive from each process
- `dtype`: data type of elements in the buffer `buf` and `recv_buf`
- `attr`: optional attributes that customize operation

> template<class **buffer_container_type**, class = *typename* std::enable_if<ccl::is_class_supported<buffer_container_type>()>::type>
> coll_request_t**alltoall**(*const* buffer_container_type &*send_buf*, buffer_container_type &*recv_buf*, size_t *count*, *const* ccl::coll_attr **attr* = nullptr, *const* ccl::stream_t &*stream* = ccl::stream_t())

Each process sends distinct data to each of the receivers. The j-th block sent from process i is received by process j and is placed in the i-th block of recvbuf.

> Return

ccl::request object that can be used to track the progress of the operation

> Parameters

- `send_buf`: the buffer with `count` elements of `dtype` that stores local data
- `recv_buf`: [out] - the buffer to store received result , must have the N * dimension of `buf`, where N - communicator size.
- `count`: number of elements to send / receive from each process
- `dtype`: data type of elements in the buffer `buf` and `recv_buf`
- `attr`: optional attributes that customize operation

> coll_request_t**alltoallv**(*const* void **send_buf*, *const* size_t **send_counts*, void **recv_buf*, *const* size_t **recv_counts*, ccl::datatype*dtype*, *const* ccl::coll_attr **attr* = nullptr, *const* ccl::stream_t &*stream* = ccl::stream_t())

Each process sends distinct data to each of the receivers. The j-th block sent from process i is received by process j and is placed in the i-th block of recvbuf. Block sizes may differ.

> Return

ccl::communicator::coll_request_t object that can be used to track the progress of the operation

Parameters

- `send_buf`: the buffer with elements of `dtype` that stores local data
- `send_counts`: array with number of elements send to each process
- `recv_buf`: [out] the buffer to store received result from each process
- `recv_counts`: array with number of elements received from each process
- `dtype`: data type of elements in the buffer `send_buf` and `recv_buf`
- `attr`: optional attributes that customize operation

template<class **buffer_type**, class = *typename* std::enable_if<ccl::is_native_type_supported<buffer_type>()>::type> coll_request_t**alltoallv**(*const* buffer_type *send_buf, *const* size_t *send_counts, buffer_type *recv_buf, *const* size_t *recv_counts, *const* ccl::coll_attr *attr = nullptr, *const* ccl::stream_t &*stream* = ccl::stream_t())

Type safety version: Each process sends distinct data to each of the receivers. The j-th block sent from process i is received by process j and is placed in the i-th block of recvbuf. Block sizes may differ.

Return

ccl::communicator::coll_request_t object that can be used to track the progress of the operation

Parameters

- `send_buf`: the buffer with elements of `dtype` that stores local data
- `send_counts`: array with number of elements send to each process
- `recv_buf`: [out] the buffer to store received result from each process
- `recv_counts`: array with number of elements received from each process
- `attr`: optional attributes that customize operation

template<class **buffer_container_type**, class = *typename* std::enable_if<ccl::is_class_supported<buffer_container_type>()>::type> coll_request_t**alltoallv**(*const* buffer_container_type &*send_buf*, *const* size_t *send_counts, buffer_container_type &*recv_buf*, *const* size_t *recv_counts, *const* ccl::coll_attr *attr = nullptr, *const* ccl::stream_t &*stream* = ccl::stream_t())

Type safety version: Each process sends distinct data to each of the receivers. The j-th block sent from process i is received by process j and is placed in the i-th block of recvbuf. Block sizes may differ.

Return

ccl::communicator::coll_request_t object that can be used to track the progress of the operation

Parameters

- `send_buf` : the buffer with elements of `dtype` that stores local data
- `send_counts` : array with number of elements send to each process
- `recv_buf` : [out] the buffer to store received result from each process
- `recv_counts` : array with number of elements received from each process
- `attr` : optional attributes that customize operation

void **barrier**(*const* ccl::stream_t &*stream* = ccl::stream_t())

Collective operation that blocks each process until every process have reached it

coll_request_t**bcast**(void **buf*, size_t *count*, ccl::datatype*dtype*, size_t *root*, *const* ccl::coll_attr **attr* = nullptr, *const* ccl::stream_t &*stream* = ccl::stream_t())

Broadcasts `buf` from the `root` process to other processes in a communicator

Return

ccl::communicator::coll_request_t object that can be used to track the progress of the operation

Parameters

- `buf` : [in,out] the buffer with `count` elements of `dtype` to be transmitted if the rank of the communicator is equal to `root` or to be received by other ranks
- `count` : number of elements of type `dtype` in `buf`
- `dtype` : data type of elements in the buffer `buf`
- `root` : the rank of the process that will transmit `buf`
- `attr` : optional attributes that customize operation

template<class **buffer_type**, class = *typename* std::enable_if<ccl::is_native_type_supported<buffer_type>()>::type>
coll_request_t**bcast**(buffer_type **buf*, size_t *count*, size_t *root*, *const* ccl::coll_attr **attr* = nullptr, *const* ccl::stream_t &*stream* = ccl::stream_t())

Type safety version: Broadcasts `buf` from the `root` process to other processes in a communicator

Return

coll_request_t object that can be used to track the progress of the operation

> **Parameters**

- `buf` : [in,out] the buffer with `count` elements of `buffer_type` to be transmitted if the rank of the communicator is equal to `root` or to be received by other ranks
- `count` : number of elements of type `buffer_type` in `buf`
- `root` : the rank of the process that will transmit `buf`
- `attr` : optional attributes that customize operation

> template<class **buffer_container_type**, class = *typename* std::enable_if<ccl::is_class_supported<buffer_container_type>()>::type>
> coll_request_t**bcast**(buffer_container_type &*buf*, size_t *count*, size_t *root*, *const* ccl::coll_attr **attr* = nullptr, *const* ccl::stream_t &*stream* = ccl::stream_t())

Type safety version: Broadcasts `buf` from the `root` process to other processes in a communicator

> **Return**

coll_request_t object that can be used to track the progress of the operation

> **Parameters**

- `buf` : [in,out] the buffer of `buffer_container_type` with `count` elements to be transmitted if the rank of the communicator is equal to `root` or to be received by other ranks
- `count` : number of elements in `buf`
- `root` : the rank of the process that will transmit `buf`
- `attr` : optional attributes that customize operation

> coll_request_t**reduce**(*const* void **send_buf*, void **recv_buf*, size_t *count*, ccl::datatype*dtype*, ccl::reduction*reduction*, size_t *root*, *const* ccl::coll_attr **attr* = nullptr, *const* ccl::stream_t &*stream* = ccl::stream_t())

Reduces `buf` on all process in the communicator and stores result in `recv_buf` on the `root` process

> **Return**

coll_request_t object that can be used to track the progress of the operation

> **Parameters**

- `send_buf` : the buffer with `count` elements of `dtype` that stores local data to be reduced

- `recv_buf`: [out] the buffer to store reduced result on the `root` process, must have the same dimension as `buf`. Used by the `root` process only, ignored by other processes
- `count`: number of elements of type `dtype` in `buf`
- `dtype`: data type of elements in the buffer `buf` and `recv_buf`
- `reduction`: type of reduction operation to be applied
- `root`: the rank of the process that will held result of reduction
- `attr`: optional attributes that customize operation

> template<class **buffer_type**, class = *typename* std::enable_if<ccl::is_native_type_supported<buffer_type>()>::type>
> coll_request_t**reduce**(*const* buffer_type **send_buf*, buffer_type **recv_buf*, size_t *count*, ccl::reduction*reduction*, size_t *root*, *const* ccl::coll_attr **attr* = nullptr, *const* ccl::stream_t &*stream* = ccl::stream_t())

Type safety version: Reduces `buf` on all process in the communicator and stores result in `recv_buf` on the `root` process

Return

ccl::communicator::coll_request_t object that can be used to track the progress of the operation

Parameters

- `send_buf`: the buffer with `count` elements of `buffer_type` that stores local data to be reduced
- `recv_buf`: [out] the buffer to store reduced result on the `root` process, must have the same dimension as `buf`. Used by the `root` process only, ignored by other processes
- `count`: number of elements of type `buffer_type` in `buf`
- `reduction`: type of reduction operation to be applied
- `root`: the rank of the process that will held result of reduction
- `attr`: optional attributes that customize operation

> template<class **buffer_container_type**, class = *typename* std::enable_if<ccl::is_class_supported<buffer_container_type>()>::type>
> coll_request_t**reduce**(*const* buffer_container_type &*send_buf*, buffer_container_type &*recv_buf*, size_t *count*, ccl::reduction*reduction*, size_t *root*, *const* ccl::coll_attr **attr* = nullptr, *const* ccl::stream_t &*stream* = ccl::stream_t())

Type safety version: Reduces `buf` on all process in the communicator and stores result in `recv_buf` on the `root` process

Return

ccl::communicator::coll_request_t object that can be used to track the progress of the operation

Parameters

- `send_buf`: the buffer of `buffer_container_type` with `count` elements that stores local data to be reduced
- `recv_buf`: [out] the buffer of `buffer_container_type` to store reduced result on the `root` process, must have the same dimension as `buf`. Used by the `root` process only, ignored by other processes
- `count`: number of elements of type `buffer_type` in `buf`
- `reduction`: type of reduction operation to be applied
- `root`: the rank of the process that will held result of reduction
- `attr`: optional attributes that customize operation

coll_request_t**sparse_allreduce**(*const* void **send_ind_buf*, size_t *send_ind_count*, *const* void **send_val_buf*, size_t *send_val_count*, void **recv_ind_buf*, size_t *recv_ind_count*, void **recv_val_buf*, size_t *recv_val_count*, ccl::datatype*index_dtype*, ccl::datatype*value_dtype*, ccl::reduction*reduction*, *const* ccl::coll_attr **attr* = nullptr, *const* ccl::stream_t &*stream* = ccl::stream_t())

Reduces sparse `buf` on all process in the communicator and stores result in `recv_buf` on each process

Return

ccl::communicator::coll_request_t object that can be used to track the progress of the operation

Parameters

- `send_ind_buf`: the buffer of indices with `send_ind_count` elements of `index_dtype`
- `send_int_count`: number of elements of type `index_type` `send_ind_buf`
- `send_val_buf`: the buffer of values with `send_val_count` elements of `value_dtype`
- `send_val_count`: number of elements of type `value_type` `send_val_buf`
- `recv_ind_buf`: [out] the buffer to store reduced indices, must have the same dimension as `send_ind_buf`

- `recv_ind_count`: [out] the amount of reduced indices
- `recv_val_buf`: [out] the buffer to store reduced values, must have the same dimension as `send_val_buf`
- `recv_val_count`: [out] the amount of reduced values
- `index_dtype`: index type of elements in the buffer `send_ind_buf` and `recv_ind_buf`
- `value_dtype`: data type of elements in the buffer `send_val_buf` and `recv_val_buf`
- `reduction`: type of reduction operation to be applied
- `attr`: optional attributes that customize operation

> template<class **index_buffer_type**, class **value_buffer_type**, class = *typename* std::enable_if<ccl::is_native_type_supported<value_buffer_type>()>::type>
> coll_request_t**sparse_allreduce**(*const* index_buffer_type **send_ind_buf*, size_t *send_ind_count*, *const* value_buffer_type **send_val_buf*, size_t *send_val_count*, index_buffer_type **recv_ind_buf*, size_t *recv_ind_count*, value_buffer_type **recv_val_buf*, size_t *recv_val_count*, ccl::reduction*reduction*, *const* ccl::coll_attr **attr* = nullptr, *const* ccl::stream_t &*stream* = ccl::stream_t())

Type safety version: Reduces sparse `buf` on all process in the communicator and stores result in `recv_buf` on each process

Return

ccl::request object that can be used to track the progress of the operation

Parameters

- `send_ind_buf`: the buffer of indices with `send_ind_count` elements of `index_buffer_type`
- `send_int_count`: number of elements of type `index_buffer_type` `send_ind_buf`
- `send_val_buf`: the buffer of values with `send_val_count` elements of `value_buffer_type`
- `send_val_count`: number of elements of type `value_buffer_type` `send_val_buf`
- `recv_ind_buf`: [out] the buffer to store reduced indices, must have the same dimension as `send_ind_buf`
- `recv_ind_count`: [out] the amount of reduced indices
- `recv_val_buf`: [out] the buffer to store reduced values, must have the same dimension as `send_val_buf`
- `recv_val_count`: [out] the amount of reduced values

- `reduction` : type of reduction operation to be applied
- `attr` : optional attributes that customize operation

> template<class **index_buffer_container_type**, class **value_buffer_container_type**, class = *typename* std::enable_if<ccl::is_class_supported<value_buffer_container_type>()>::type>
> coll_request_t**sparse_allreduce**(*const* index_buffer_container_type &*send_ind_buf*, size_t *send_ind_count*, *const* value_buffer_container_type &*send_val_buf*, size_t *send_val_count*, index_buffer_container_type &*recv_ind_buf*, size_t *recv_ind_count*, value_buffer_container_type &*recv_val_buf*, size_t *recv_val_count*, ccl::reduction*reduction*, *const* ccl::coll_attr **attr* = nullptr, *const* ccl::stream_t &*stream* = ccl::stream_t())

Type safety version: Reduces sparse `buf` on all process in the communicator and stores result in `recv_buf` on each process

Return

ccl::request object that can be used to track the progress of the operation

Parameters

- `send_ind_buf` : the buffer of `index_buffer_container_type` of indices with `send_ind_count` elements
- `send_int_count` : number of elements of `send_ind_buf`
- `send_val_buf` : the buffer of `value_buffer_container_type` of values with `send_val_count` elements
- `send_val_count` : number of elements of in `send_val_buf`
- `recv_ind_buf` : [out] the buffer of `index_buffer_container_type` to store reduced indices, must have the same dimension as `send_ind_buf`
- `recv_ind_count` : [out] the amount of reduced indices
- `recv_val_buf` : [out] the buffer of `value_buffer_container_type` to store reduced values, must have the same dimension as `send_val_buf`
- `recv_val_count` : [out] the amount of reduced values
- `reduction` : type of reduction operation to be applied
- `attr` : optional attributes that customize operation

Docs » Library API » Class environment

Class environment

- Defined in File ccl.hpp

Class Documentation

class **`environment`**

ccl environment singleton

Public Functions

`~environment`()

void **`set_resize_fn`**(ccl_resize_fn_t*callback*)

Enables job scalability policy

Parameters

- `callback`: of `ccl_resize_fn_t` type, which enables scalability policy (`nullptr` enables default behavior)

communicator_t**`create_communicator`**(*const* ccl::comm_attr_t &*attr* = ccl::comm_attr_t())*const*

Creates a new communicator according to `attr` parameters or creates a copy of global communicator, if `attr` is `nullptr(default)`

Parameters

- `attr`:

template<class **`stream_native_type`**, class = *typename* std::enable_if<is_stream_supported<stream_native_type>()>::type>
stream_t**`create_stream`**(stream_native_type &*native_stream*)

Creates a new ccl stream from @stream_native_type

Parameters

- `native_stream`: the existing handle of stream

stream_t**create_stream**()*const*

ccl_version_t**get_version**()*const*

Retrieves the current version

comm_attr_t**create_host_comm_attr**(*const* ccl_host_comm_attr_t &*attr* = ccl_host_comm_attr_t())*const*

Created @attr, which used to create host from @environment

Public Static Functions

static environment &**instance**()

Retrieves the unique ccl environment object and makes the first-time initialization of ccl library

Class request

- Defined in File ccl.hpp

Class Documentation

class **request**

A request interface that allows the user to track collective operation progress

Public Functions

virtual void **wait**() = 0

Blocking wait for collective operation completion

virtual bool **test**() = 0

Non-blocking check for collective operation completion

Return Value

- `true` : if the operations has been completed
- `false` : if the operations has not been completed

virtual **~request**()

Class stream

- Defined in File ccl.hpp

Class Documentation

class **stream**

A stream object is an abstraction over CPU/GPU streams Has no defined public constructor. Use ccl::environment::create_stream for stream objects creation

Public Types

using **impl_t** = std::shared_ptr<ccl_stream>

Public Functions

stream(*const* stream&)

stream is not copyable

stream &**operator=**(*const* stream&)

stream(stream&&)

stream is movable

stream &**operator=**(stream&&)

Enum datatype

- Defined in File ccl_types.hpp

Enum Documentation

enum **`ccl::datatype`**

Supported datatypes

Values:

`dt_char` = ccl_dtype_char

`dt_int` = ccl_dtype_int

`dt_bfp16` = ccl_dtype_bfp16

`dt_float` = ccl_dtype_float

`dt_double` = ccl_dtype_double

`dt_int64` = ccl_dtype_int64

`dt_uint64` = ccl_dtype_uint64

`dt_last_value` = ccl_dtype_last_value

Enum reduction

- Defined in File ccl_types.hpp

Enum Documentation

enum **`ccl::reduction`**

Supported reduction operations

Values:

`sum` = ccl_reduction_sum

`prod` = ccl_reduction_prod

`min` = ccl_reduction_min

`max` = ccl_reduction_max

`custom` = ccl_reduction_custom

`last_value` = ccl_reduction_last_value

Enum stream_type

- Defined in File ccl_types.hpp

Enum Documentation

enum **`ccl::stream_type`**

Supported stream types

Values:

> **`host`** = ccl_stream_host

> **`cpu`** = ccl_stream_cpu

> **`gpu`** = ccl_stream_gpu

> **`last_value`** = ccl_stream_last_value

Enum ccl_host_attributes

- Defined in File ccl_types.h

Enum Documentation

enum **`ccl_host_attributes`**

Host attributes

Values:

> **`ccl_host_color`**

> **`ccl_host_version`**

Enum ccl_reduction_t

- Defined in File ccl_types.h

Enum Documentation

enum **`ccl_reduction_t`**

Reduction operations.

Values:

> **`ccl_reduction_sum`** = 0

> **`ccl_reduction_prod`** = 1

> **`ccl_reduction_min`** = 2

> **`ccl_reduction_max`** = 3

> **`ccl_reduction_custom`** = 4

> **`ccl_reduction_last_value`**

Enum ccl_resize_action

- Defined in File ccl_types.h

Enum Documentation

enum **`ccl_resize_action`**

Resize action types.

Values:

> **`ccl_ra_wait`** = 0

> **`ccl_ra_run`** = 1

> **`ccl_ra_finalize`** = 2

Enum ccl_status_t

- Defined in File ccl_types.h

Enum Documentation

enum **`ccl_status_t`**

Status values returned by CCL functions.

Values:

`ccl_status_success` = 0

`ccl_status_out_of_resource` = 1

`ccl_status_invalid_arguments` = 2

`ccl_status_unimplemented` = 3

`ccl_status_runtime_error` = 4

`ccl_status_blocked_due_to_resize` = 5

`ccl_status_last_value`

Enum ccl_stream_type_t

- Defined in File ccl_types.h

Enum Documentation

enum **`ccl_stream_type_t`**

Stream types.

Values:

`ccl_stream_host` = 0

`ccl_stream_cpu` = 1

`ccl_stream_gpu` = 2

`ccl_stream_last_value`