Intel® oneAPI Collective Communications Library Developer Guide

June 25, 2021

Release 2021.3

Contents

List of Figures

List of Tables

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 / distributed frameworks (such as PyTorch*, Horovod*).

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

1.0 Release Notes

Refer to Intel® oneAPI Collective Communications Library Release Notes.

2.0 Installation Guide

This page explains how to install and configure the Intel® oneAPI Collective Communications Library (oneCCL). oneCCL supports different installation scenarios using command line interface.

2.1 System Requirements

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

2.2 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.

2.2.1 Custom Installation

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

- To specify **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=<c_compiler> -DCMAKE_CXX_COMPILER=<cxx_compiler>
```

- To enable `SYCL` devices communication support, specify `SYCL` compiler and set `-DCOMPUTE_BACKEND` (only DPC++ is supported):

```
cmake .. -DCMAKE_C_COMPILER=clang -DCMAKE_CXX_COMPILER=dpcpp -DCOMPUTE_BACKEND=dpcpp
```

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

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

- 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 install
```

2.3 Environment Setup

Before you start using oneCCL, make sure to set up the library environment. There are two ways to set up the environment:

- Using standalone oneCCL package installed into `<ccl_install_dir>`:

```
source <ccl_install_dir>/setvars.sh
```

- Using oneCCL from Intel® oneAPI Base Toolkit installed into `<toolkit_install_dir>` (`/opt/intel/inteloneapi` by default):

```
source <toolkit_install_dir>/setvars.sh
```

3.0 Sample Application

The sample code below shows how to use oneCCL API to perform allreduce communication for SYCL* memory: buffer objects and USM.

SYCL Buffers

```
#include "sycl_base.hpp"

using namespace std;
using namespace sycl;

int main(int argc, char *argv[]) {
    const size_t count = 10 * 1024 * 1024;

    int i = 0;
    int size = 0;
    int rank = 0;

    ccl::init();

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    atexit(mpi_finalize);

    queue q;
    if (!create_sycl_queue(argc, argv, rank, q)) {
        return -1;
    }

    /* create kvs */
    ccl::shared_ptr_class<ccl::kvs> kvs;
    ccl::kvs::address_type main_addr;
    if (rank == 0) {
        kvs = ccl::create_main_kvs();
        main_addr = kvs->get_address();
        MPI_Bcast((void *)main_addr.data(), main_addr.size(), MPI_BYTE, 0, MPI_COMM_WORLD);
    }
    else {
        MPI_Bcast((void *)main_addr.data(), main_addr.size(), MPI_BYTE, 0, MPI_COMM_WORLD);
        kvs = ccl::create_kvs(main_addr);
    }

    /* create communicator */
    auto dev = ccl::create_device(q.get_device());
    auto ctx = ccl::create_context(q.get_context());
    auto comm = ccl::create_communicator(size, rank, dev, ctx, kvs);
```

(continues on next page)

(continued from previous page)

```
    /* create stream */
    auto stream = ccl::create_stream(q);

    /* create buffers */
    buffer<int> send_buf(count);
    buffer<int> recv_buf(count);

    {
        /* open buffers and initialize them on the host side */
        host_accessor send_buf_acc(send_buf, write_only);
        host_accessor recv_buf_acc(recv_buf, write_only);
        for (i = 0; i < count; i++) {
            send_buf_acc[i] = rank;
            recv_buf_acc[i] = -1;
        }
    }

    /* open send_buf and modify it on the device side */
    q.submit([&](auto &h) {
        accessor send_buf_acc(send_buf, h, write_only);
        h.parallel_for(count, [=](auto id) {
            send_buf_acc[id] += 1;
        });
    });

    if (!handle_exception(q))
        return -1;

    /* invoke allreduce */
    ccl::allreduce(send_buf, recv_buf, count, ccl::reduction::sum, comm, stream).wait();

    /* open recv_buf and check its correctness on the device side */
    q.submit([&](auto &h) {
        accessor recv_buf_acc(recv_buf, h, write_only);
        h.parallel_for(count, [=](auto id) {
            if (recv_buf_acc[id] != size * (size + 1) / 2) {
                recv_buf_acc[id] = -1;
            }
        });
    });

    if (!handle_exception(q))
        return -1;

    /* print out the result of the test on the host side */
    {
        host_accessor recv_buf_acc(recv_buf, read_only);
        for (i = 0; i < count; i++) {
            if (recv_buf_acc[i] == -1) {
                cout << "FAILED\n";
```

(continues on next page)

(continued from previous page)

```
                break;
            }
        }
        if (i == count) {
            cout << "PASSED\n";
        }
    }

    return 0;
}
```

SYCL USM

```
#include "sycl_base.hpp"

using namespace std;
using namespace sycl;

int main(int argc, char *argv[]) {
    const size_t count = 10 * 1024 * 1024;

    int i = 0;
    int size = 0;
    int rank = 0;

    ccl::init();

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    atexit(mpi_finalize);

    queue q;
    if (!create_sycl_queue(argc, argv, rank, q)) {
        return -1;
    }

    buf_allocator<int> allocator(q);

    auto usm_alloc_type = usm::alloc::shared;
    if (argc > 2) {
        usm_alloc_type = usm_alloc_type_from_string(argv[2]);
    }

    if (!check_sycl_usm(q, usm_alloc_type)) {
        return -1;
    }

    /* create kvs */
    ccl::shared_ptr_class<ccl::kvs> kvs;
```

(continues on next page)

(continued from previous page)

```
    ccl::kvs::address_type main_addr;
    if (rank == 0) {
        kvs = ccl::create_main_kvs();
        main_addr = kvs->get_address();
        MPI_Bcast((void *)main_addr.data(), main_addr.size(), MPI_BYTE, 0, MPI_COMM_WORLD);
    }
    else {
        MPI_Bcast((void *)main_addr.data(), main_addr.size(), MPI_BYTE, 0, MPI_COMM_WORLD);
        kvs = ccl::create_kvs(main_addr);
    }

    /* create communicator */
    auto dev = ccl::create_device(q.get_device());
    auto ctx = ccl::create_context(q.get_context());
    auto comm = ccl::create_communicator(size, rank, dev, ctx, kvs);

    /* create stream */
    auto stream = ccl::create_stream(q);

    /* create buffers */
    auto send_buf = allocator.allocate(count, usm_alloc_type);
    auto recv_buf = allocator.allocate(count, usm_alloc_type);

    /* open buffers and modify them on the device side */
    auto e = q.submit([&](auto &h) {
        h.parallel_for(count, [=](auto id) {
            send_buf[id] = rank + 1;
            recv_buf[id] = -1;
        });
    });

    if (!handle_exception(q))
        return -1;

    /* invoke allreduce */
    ccl::allreduce(send_buf, recv_buf, count, ccl::reduction::sum, comm, stream).wait();

    /* open recv_buf and check its correctness on the device side */
    buffer<int> check_buf(count);
    q.submit([&](auto &h) {
        accessor check_buf_acc(check_buf, h, write_only);
        h.parallel_for(count, [=](auto id) {
            if (recv_buf[id] != size * (size + 1) / 2) {
                check_buf_acc[id] = -1;
            }
        });
    });

    if (!handle_exception(q))
        return -1;
```

(continues on next page)

(continued from previous page)

```
    /* print out the result of the test on the host side */
    {
        host_accessor check_buf_acc(check_buf, read_only);
        for (i = 0; i < count; i++) {
            if (check_buf_acc[i] == -1) {
                cout << "FAILED\n";
                break;
            }
        }
        if (i == count) {
            cout << "PASSED\n";
        }
    }

    return 0;
}
```

3.1 Build details

1. Build oneCCL with SYCL support (only DPC++ is supported).
2. Set up the library environment.
3. Use clang++ compiler to build the sample:

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

3.2 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> ./sample
```

where <parameters> represents optional mpiexec parameters such as node count, processes per node, hosts, and so on.

4.0 Programming Model

4.1 Host Communication

The communication operations between processes are provided by Communicator.

The example below demonstrates the main concepts of communication on host memory buffers.

Example

Consider a simple oneCCL `allreduce` example for CPU.

1. Create a communicator object with user-supplied size, rank, and key-value store:

```
auto ccl_context = ccl::create_context();
auto ccl_device = ccl::create_device();

auto comms = ccl::create_communicators(
   size,
   vector_class<pair_class<size_t, device>>{ { rank, ccl_device } },
   ccl_context,
   kvs);
```

   Or for convenience use non-vector form without device and context parameters.

```
auto comm = ccl::create_communicator(size, rank, kvs);
```

2. Initialize `send_buf` (in real scenario it is supplied by the user):

```
const size_t elem_count = <N>;

/* initialize send_buf */
for (idx = 0; idx < elem_count; idx++) {
   send_buf[idx] = rank + 1;
}
```

3. `allreduce` invocation performs the reduction of values from all the processes and then distributes the result to all the processes. In this case, the result is an array with `elem_count` elements, where all elements are equal to the sum of arithmetical progression:

$$p \cdot (p+1)/2$$

```
ccl::allreduce(send_buf,
               recv_buf,
               elem_count,
               reduction::sum,
               comm).wait();
```

4. Check the correctness of `allreduce` operation:

```
auto comm_size = comm.size();
auto expected = comm_size * (comm_size + 1) / 2;

for (idx = 0; idx < elem_count; idx++) {
   if (recv_buf[idx] != expected) {
         std::count << "unexpected value at index " << idx << std::endl;
         break;
   }
}
```

4.2 Device Communication

The communication operations between devices are provided by Communicator.

The example below demonstrates the main concepts of communication on device memory buffers.

Example

Consider a simple oneCCL `allreduce` example for GPU:

1. Create oneCCL communicator objects with user-supplied size, rank <-> SYCL device mapping, SYCL context and key-value store:

```
auto ccl_context = ccl::create_context(sycl_context);
auto ccl_device = ccl::create_device(sycl_device);

auto comms = ccl::create_communicators(
   size,
   vector_class<pair_class<size_t, device>>{ { rank, ccl_device } },
   ccl_context,
   kvs);
```

2. Create oneCCL stream object from user-supplied `sycl::queue` object:

```
auto stream = ccl::create_stream(sycl_queue);
```

3. Initialize `send_buf` (in real scenario it is supplied by the user):

```
const size_t elem_count = <N>;

/* using SYCL buffer and accessor */
auto send_buf_host_acc = send_buf.get_access<mode::write>();
for (idx = 0; idx < elem_count; idx++) {
   send_buf_host_acc[idx] = rank;
}
```

```
/* or using SYCL USM */
for (idx = 0; idx < elem_count; idx++) {
   send_buf[idx] = rank;
}
```

4. For demonstration purposes, modify the `send_buf` on the GPU side:

```
/* using SYCL buffer and accessor */
sycl_queue.submit([&](cl::sycl::handler& h) {
   auto send_buf_dev_acc = send_buf.get_access<mode::write>(h);
   h.parallel_for(range<1>{elem_count}, [=](item<1> idx) {
         send_buf_dev_acc[idx] += 1;
   });
});
```

```
/* or using SYCL USM */
for (idx = 0; idx < elem_count; idx++) {
   send_buf[idx]+ = 1;
}
```

5. `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 `elem_count` elements, where all elements are equal to the sum of arithmetical progression:

$$p \cdot (p+1)/2$$

```
std::vector<event> events;
for (auto& comm : comms) {
   events.push_back(ccl::allreduce(send_buf,
                                   recv_buf,
                                   elem_count,
                                   reduction::sum,
                                   comm,
                                   streams[comm.rank()]));
}

for (auto& e : events) {
   e.wait();
}
```

6. Check the correctness of `allreduce` operation on the GPU:

```
/* using SYCL buffer and accessor */

auto comm_size = comm.size();
auto expected = comm_size * (comm_size + 1) / 2;

sycl_queue.submit([&](handler& h) {
   auto recv_buf_dev_acc = recv_buf.get_access<mode::write>(h);
   h.parallel_for(range<1>{elem_count}, [=](item<1> idx) {
         if (recv_buf_dev_acc[idx] != expected) {
            recv_buf_dev_acc[idx] = -1;
         }
   });
});
```

(continues on next page)

(continued from previous page)

```
...

auto recv_buf_host_acc = recv_buf.get_access<mode::read>();
for (idx = 0; idx < elem_count; idx++) {
   if (recv_buf_host_acc[idx] == -1) {
         std::count << "unexpected value at index " << idx << std::endl;
         break;
   }
}
```

```
/* or using SYCL USM */

auto comm_size = comm.size();
auto expected = comm_size * (comm_size + 1) / 2;

for (idx = 0; idx < elem_count; idx++) {
   if (recv_buf[idx] != expected) {
         std::count << "unexpected value at index " << idx << std::endl;
         break;
   }
}
```

4.3 Limitations

The list of scenarios not yet supported by oneCCL:

- Creation of multiple ranks within single process
- Handling of dependencies as operation parameter (for example, `deps` vector in `ccl::allreduce(..., deps)`)
- Float16 datatype support

See also:

Check out oneCCL specification that oneCCL is based on.

5.0 General Configuration

5.1 Execution of Communication Operations

Communication 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 by CCL workers. The general rule of thumb is to use different CPU cores for compute (e.g. by specifying `KMP_AFFINITY`) and for CCL communication.

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

5.1.1 Automatic setup

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

Example

In the example below, oneCCL creates four workers per process and pins them to the last four cores available for the process (available if `mpirun` launcher from oneCCL package is used, the exact IDs of CPU cores depend on the parameters passed to `mpirun`) or to the last four cores on the node.

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

5.1.2 Explicit setup

To set affinity explicitly for all local workers, pass ID of the cores to the `CCL_WORKER_AFFINITY` environment variable.

Example

In the example below, oneCCL creates 4 workers per process 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
```

5.2 Transport selection

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

The transport selection is controlled by **CCL_ATL_TRANSPORT**.

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

6.0 Advanced Configuration

6.1 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.

6.2 Caching of Communication Operations

Communication 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 operation internal representations and reusing them on the subsequent calls.

To control this, use operation attribute and set `true` value for `to_cache` field and unique string (for example, tensor name) for `match_id` field.

Note that:

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

6.3 Prioritization of Communication Operations

oneCCL supports prioritization of communication operations that controls the order in which individual communication operations are executed. This allows to postpone execution of non-urgent operations to complete urgent operations earlier, which may be beneficial for many use cases.

The communication prioritization is controlled by priority value. Note that the priority must be a non-negative number with a higher number standing for a higher priority.

There are the following prioritization modes:

- None - default mode when all communication operations have the same priority.
- Direct - you explicitly specify priority using `priority` field in operation attribute.
- LIFO (Last In, First Out) - priority is implicitly increased on each operation call. In this case, you do not have to specify priority.

The prioritization mode is controlled by CCL_PRIORITY.

6.4 Fusion of Communication Operations

In some cases, it may be beneficial to postpone execution of communication 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

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

6.5 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 string identifier. To set an identifier, use `match_id` field in operation attribute.

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`.

Unordered collectives are controlled by CCL_UNORDERED_COLL.

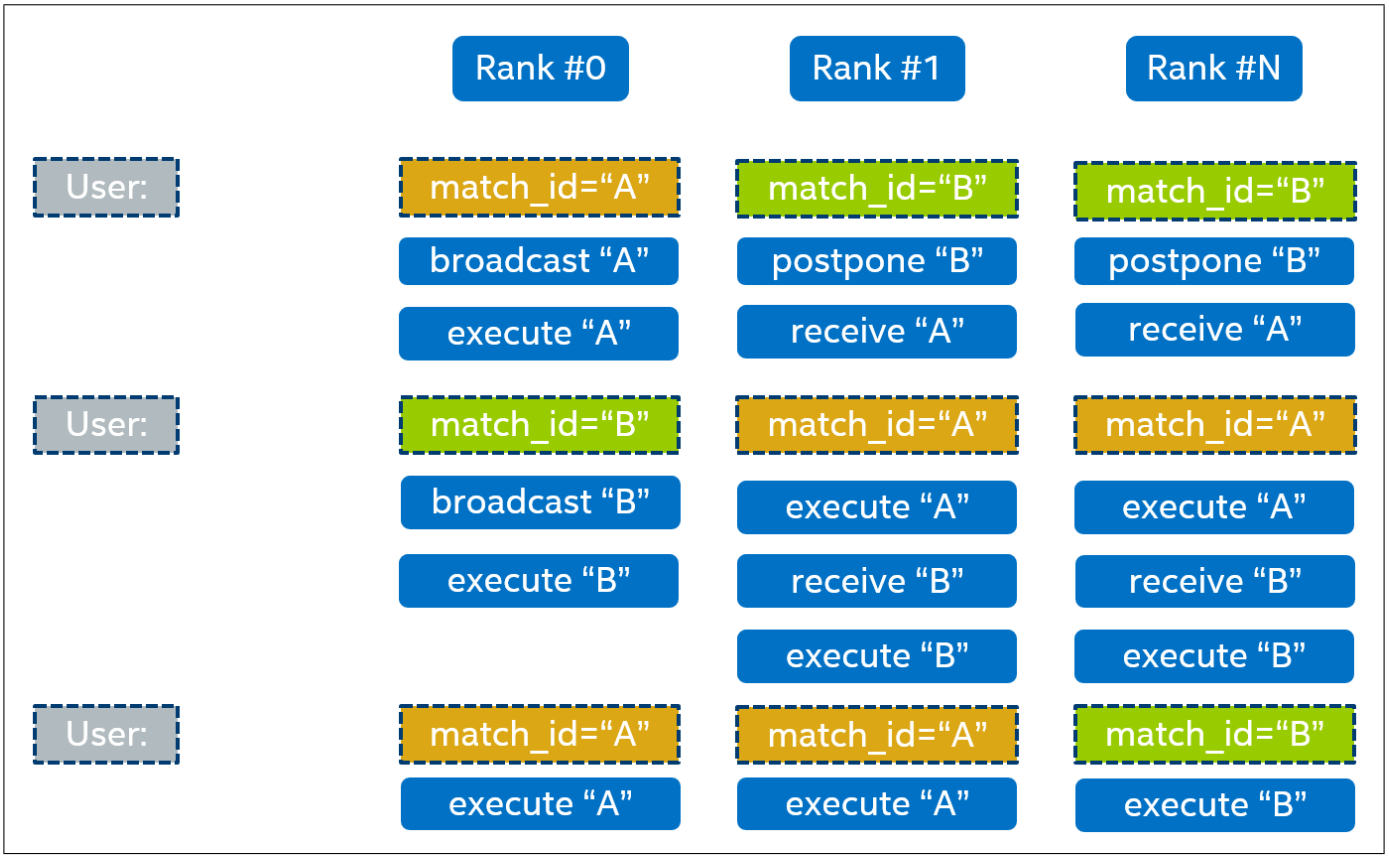

Fig. 1: Unordered collective operations

6.6 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.

Sparse allreduce is a collective communication operation that makes global reduction operation on sparse buffers from all ranks of communicator and distributes result back to all ranks. Sparse buffers are defined by separate index and value buffers.

```
ccl::event 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 ind_dtype,
                            ccl::datatype val_dtype,
                            ccl::reduction rtype,
                            const ccl::communicator& comm,
                            const ccl::stream& stream,
                            const ccl::sparse_allreduce_attr& attr = ccl::default_sparse_
↪allreduce_attr,
                            const ccl::vector_class<ccl::event>& deps = {});
```

send_ind_buf the buffer of indices with `send_ind_count` elements of type `ind_dtype`

send_ind_count the number of elements of type `ind_type` in `send_ind_buf`

send_val_buf the buffer of values with `send_val_count` elements of type `val_dtype`

send_val_count the number of elements of type `val_type` in `send_val_buf`

recv_ind_buf [out] the buffer to store reduced indices, unused

recv_ind_count [out] the number of elements in `recv_ind_buf`, unused

recv_val_buf [out] the buffer to store reduced values, unused

recv_val_count [out] the number of elements in `recv_val_buf`, unused

ind_dtype the datatype of elements in `send_ind_buf` and `recv_ind_buf`

val_dtype the the datatype of elements in `send_val_buf` and `recv_val_buf`

rtype the type of the reduction operation to be applied

comm the communicator that defines a group of ranks for the operation

stream an optional stream associated with the operation

attr optional attributes to customize the operation

deps an optional vector of the events that the operation should depend on

return `event` an object to track the progress of the operation

For `sparse_allreduce`, a completion callback or an allocation callback is required.

Use the following fields in operation attribute:

- `completion_fn` - a completion callback function pointer
- `alloc_fn` - an allocation callback function pointer
- `fn_ctx` - an user context pointer of type `void*`

Completion callback should follow the signature:

```
typedef void (*completion_fn)
(
    const void*,   /* idx_buf      */
    size_t,        /* idx_count    */
    ccl::datatype, /* idx_dtype    */
    const void*,   /* val_buf      */
    size_t,        /* val_count    */
    ccl::datatype, /* val_dtype    */
    const void*    /* user_context */
);
```

Note that `idx_buf` and `val_buf` are temporary buffers. Thus, the data from these buffers should be copied. Use `user_context` for this purpose.

Allocation callback should follow the signature:

```
typedef void (*alloc_fn)
(
    size_t,        /* idx_count    */
    ccl::datatype, /* idx_dtype    */
    size_t,        /* val_count    */
    ccl::datatype, /* val_dtype    */
    const void*,   /* user_context */
    void**,        /* out_idx_buf  */
    void**         /* out_val_buf  */
);
```

Warning: `ccl::sparse_allreduce` is experimental and subject to change.

7.0 Environment Variables

7.1 Collective algorithms selection

7.1.1 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=info.

Example

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

7.1.2 Available collectives

Available collective operations (<coll_name>):

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

7.1.3 Available algorithms

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

ALLGATHERV algorithms

Table 1: 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

Table 2: 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_MIN_CHUNK_SIZE to control pipelining on reduce_scatter phase.
ring_rma	reduce_scatter+allgather ring using RMA communications
double_tree	Double-tree algorithm
recursive_doubling	Recursive doubling algorithm
2d	Two-dimensional algorithm (reduce_scatter + allreduce + allgather)

ALLTOALL algorithms

Table 3: ALLTOALL algorithms

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

ALLTOALLV algorithms

Table 4: ALLTOALLV algorithms

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

BARRIER algorithms

Table 5: BARRIER algorithms

direct	Based on MPI_Ibarrier
ring	Ring-based algorithm

BCAST algorithms

Table 6: BCAST algorithms

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

REDUCE algorithms

Table 7: REDUCE algorithms

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

REDUCE_SCATTER algorithms

Table 8: REDUCE_SCATTER algorithms

direct	Based on MPI_Ireduce_scatter_block
ring	Use CCL_RS_CHUNK_COUNT and CCL_RS_MIN_CHUNK_SIZE to control pipelining.

SPARSE_ALLREDUCE algorithms

Table 9: SPARSE_ALLREDUCE algorithms

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

Note: WARNING: ccl::sparse_allreduce is experimental and subject to change.

CCL_RS_CHUNK_COUNT

Syntax

```
CCL_RS_CHUNK_COUNT=<value>
```

Arguments

Table 10: CCL_RS_CHUNK_COUNT 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

Table 11: CCL_RS_MIN_CHUNK_SIZE 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`.

7.2 Fusion

7.2.1 CCL_FUSION

Syntax

```
CCL_FUSION=<value>
```

Arguments

Table 12: CCL_FUSION 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.

7.2.2 CCL_FUSION_BYTES_THRESHOLD

Syntax

```
CCL_FUSION_BYTES_THRESHOLD=<value>
```

Arguments

Table 13: CCL_FUSION_BYTES_THRESHOLD arguments

<value>	Description
SIZE	Bytes threshold for a collective operation. If the size of a communication buffer in bytes is less than or equal to SIZE, then oneCCL fuses this operation with the other ones.

Description

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

7.2.3 CCL_FUSION_COUNT_THRESHOLD

Syntax

```
CCL_FUSION_COUNT_THRESHOLD=<value>
```

Arguments

Table 14: CCL_FUSION_COUNT_THRESHOLD arguments

<value>	Description
COUNT	The threshold for the number of collective operations. oneCCL can fuse together no more than COUNT operations at a time.

Description

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

7.2.4 CCL_FUSION_CYCLE_MS

Syntax

```
CCL_FUSION_CYCLE_MS=<value>
```

Arguments

Table 15: CCL_FUSION_CYCLE_MS 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.

7.3 CCL_ATL_TRANSPORT

Syntax

```
CCL_ATL_TRANSPORT=<value>
```

Arguments

Table 16: CCL_ATL_TRANSPORT arguments

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

Description

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

7.4 CCL_UNORDERED_COLL

Syntax

```
CCL_UNORDERED_COLL=<value>
```

Arguments

Table 17: CCL_UNORDERED_COLL arguments

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

Description

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

7.5 CCL_PRIORITY

Syntax

```
CCL_PRIORITY=<value>
```

Arguments

Table 18: CCL_PRIORITY arguments

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

Description

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

7.6 CCL_WORKER_COUNT

Syntax

```
CCL_WORKER_COUNT=<value>
```

Arguments

Table 19: CCL_WORKER_COUNT 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.

7.7 CCL_WORKER_AFFINITY

Syntax

```
CCL_WORKER_AFFINITY=<proclist>
```

Arguments

Table 20: CCL_WORKER_AFFINITY arguments

<proclist>	Description
auto	Workers are automatically pinned to last cores of pin domain. Pin domain depends from process launcher. If `mpirun` from oneCCL package is used then pin domain is MPI process pin domain. Otherwise, pin domain is all cores on the node.
n1,n2,..	Affinity is explicitly specified for all local workers.

Description

Set this environment variable to specify cpu affinity for oneCCL worker threads.

7.8 CCL_LOG_LEVEL

Syntax

```
CCL_LOG_LEVEL=<value>
```

Arguments

Table 21: CCL_LOG_LEVEL arguments

<value>
error
warn (**default**)
info
debug
trace

Description

Set this environment variable to control logging level.

7.9 CCL_MAX_SHORT_SIZE

Syntax

```
CCL_MAX_SHORT_SIZE=<value>
```

Arguments

Table 22: CCL_MAX_SHORT_SIZE arguments

<value>	Description
SIZE	Bytes threshold for a collective operation (0 if not specified). If the size of a communication buffer in bytes is less than or equal to SIZE, then oneCCL does not split operation between workers. Applicable for allreduce, reduce and broadcast.

Description

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

8.0 Notices and Disclaimers

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

No product or component can be absolutely secure.

Your costs and results may vary.

Intel technologies may require enabled hardware, software or service activation.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.