CCF Transactions on High Performance Computing
https://doi.org/10.1007/s42514-025-00215-4

REGULAR PAPER

SuCL: supply unified communication layer to improve SYCL-based heterogeneous computing

Hengzhong Liang[1] · **Han Huang**[1] · **Xianwei Zhang**[1]

Received: 20 August 2024 / Accepted: 11 February 2025

Abstract
Supercomputers and data centers are continuously developing on scales and capabilities to empower scientific and intelligent applications. As the *de facto* standard to offer dense computation, various accelerators like GPUs have been widely deployed, which inevitably incurs the heterogeneous programming and usage issues. Targeting at addressing the issues, SYCL has been proposed to facilitate programs to run on different platforms based on varying accelerators and vendors. However, SYCL has a limited functionality to conduct communication between devices, so SYCL resorts to MPI or vendor-specific communication libraries, neither of which could fulfill the demand of portability and performance for SYCL programs at the same time. To overcome the dilemma of portability and performance, we propose SuCL, a communication-specific library and framework which provides an abstraction layer atop of various programming models. SuCL provides unified communication APIs for upper SYCL programs, and leverages vendor-optimized communication libraries to improve performance. To ensure program functionality, SuCL introduces selection mechanism to help selecting proper communication libraries for SYCL programs at runtime. SuCL also utilizes additional SYCL features to improve performance and programming easiness. Experiments on different platforms show that SuCL outperforms MPI in micro-benchmarks significantly, and in application evaluations SuCL is capable to produce speedups up to 60% and 30% on NVIDIA platform and AMD platform respectively.

Keywords Heterogeneous · SYCL · Portability · MPI · NCCL

1 Introduction

Modern data centers and supercomputers are increasingly reliant on heterogeneous accelerators to provide higher computational power. For supercomputers listed in Top500, such as EL Captain, Frontier and Aurora, they are equipped with abundant GPUs to bring peak performance over 1 EFLops into reality TOP500 (2024). Accelerator predominated systems are also widely implemented in emerging supercomputer internet to empower the converged high-performance and intelligent computing paradigms. As accelerators come from different vendors and may vary in functionalities and even in forms, software developers are required to utilize various programming models and libraries to fully unleash their computational potential. Thus, it becomes a difficulty to write codes able to migrate from one platform to another. To solve the problem of program portability, KHRONOS proposes SYCL[2], an open standard for heterogeneous computing. SYCL provides an abstraction layer upon various programming models as well as a set of unified programming interfaces and abstractions to harness various accelerators. With SYCL programmers could write codes portable across different platforms.

SYCL is making great effort to offer the optimal performance of a device to programmers, but as a specification focusing on computing, SYCL still lacks essential functionalities to support efficient communication among devices. Thus, to make multiple devices collaborating on computational tasks, SYCL programs resort to MPI, an industrial standard and platform-agnostic solution to communication, to manage inter-device communication. However, MPI is not a satisfying solution to communication in SYCL. MPI requires data to be sent and received on CPU, thus incurring

✉ Xianwei Zhang
zhangxw79@mail.sysu.edu.cn

Hengzhong Liang
lianghzh23@mail2.sysu.edu.cn

Han Huang
huangh367@mail2.sysu.edu.cn

[1] School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou 510006, Guangdong, China

Published online: 25 March 2025

additional data transfers between CPU and devices and further introducing performance bottleneck to SYCL programs. To avoid the bottleneck, accelerator vendors bring up optimized libraries for their devices. Vendor-specific libraries conduct direct and CPU-bypassing data exchanges between devices, thus helping boosting performance. However, using vendor-specific libraries in SYCL programs hurts program portability. While SYCL prioritizes portability and aims to enable programs to run on various platforms, vendor-specific libraries are usually bound to particular platforms and are available for certain devices only. Thus, a SYCL program working with any vendor-specific library could only run on limited platforms, which is in conflict with SYCL's prioritization of portability.

To break the dilemma of portability and performance, in this article, we propose `SuCL`, a unified communication layer for SYCL. Unlike MPI or vendor-specific libraries, `SuCL` is a library and framework originally designed to support communication for SYCL programs. `SuCL` not only provides communication functionality for SYCL, but also guarantees both program portability and performance at the same time. `SuCL` is an open framework enabling quick integration of support for new platforms. In order to ensure portability and performance at the same time, `SuCL` is built on various platforms and libraries, and adopts layered architecture to avoid getting bound to particular platforms. The upper layer of `SuCL` defines unified communication APIs for SYCL programs, and the lower part interacts with underlying libraries and provides implementations of communication routines. To avoid a SYCL program from failing to execute, `SuCL` introduces a configurable selection mechanism to help with selecting a proper communication library to be used in the program.

In summary, this article makes following contributions:

1. We point out and analyze the challenge of utilizing MPI and vendor-specific libraries with SYCL happening in data centers and supercomputers.
2. We propose `SuCL`, an open library and framework providing a unified abstraction layer to support communication in SYCL programs.
3. We implement `SuCL` backends for two programming models, CUDA and HIP, and for two SYCL implementations, Intel oneAPI and AdaptiveCPP.
4. We conduct evaluations on `SuCL` and implemented backends, and compare `SuCL` with MPI. The results show that `SuCL` outperforms MPI in both micro-benchmarks and application benchmarks, and produces performance close to the optimal one.

The arrangement of this article is as follow. In Sect. 1, we make a brief introduction of the whole article. In Sect. 2, we briefly introduce SYCL, MPI and vendor-specific library for a better understanding of the problem. In Sect. 3, we talk about the challenge of using MPI and vendor-specific libraries in SYCL programs and the motivation to propose `SuCL`. In Sect. 4, we make a detailed description of the architecture and design of `SuCL`. In Sect. 5, we provide simple code samples of basic usage of `SuCL` and implementation of a backend. In Sect. 6, we conduct evaluations on MPI and proposed `SuCL`. In Sect. 7, we discuss related works, and finally in section 8, we make a conclusion and talk about possible future work.

2 Preliminary

2.1 SYCL

SYCL is an open standard for heterogeneous computing, proposed and currently maintained by KHRONOS. The goal of SYCL is to achieve performance rival to the one that native programming model could provide on any computing platform, while at the same time lessening programming burden and ensuring program portability. Unlike CUDA or ROCm, SYCL itself is not bound to particular vendors or devices. Instead, it is built on top of other platforms and programming models as an abstract programming layer.

SYCL defines a series of abstractions and interfaces to cover the diversity of different platforms and programming models. For example, in SYCL, devices and task streams are referred to as instances of `sycl::device` and `sycl::queue` respectively. Memory management in SYCL is also different from other programming models. In SYCL, memory regions are managed using `sycl::buffer`, and should be accessed using `sycl::accessor` on devices or `sycl::host_accessor` on host. In the recent SYCL 2020 specification, memory regions can be accessed using USM, short for Unified Shared Memory, which is an optional feature though. With SYCL abstractions, programmers are able to write codes portable to any platforms.

While new abstractions are used to hide platform diversity, SYCL also introduces backends to interact with underlying platforms and programming models. A backend represents a certain platform and programming model in SYCL and provides implementations for SYCL abstractions and programming interfaces. SYCL relies on backends to capsulate and manage runtime resources, as the same kind

of resources, such as device memories, could be present in different forms on different platforms. For example, a `sycl::buffer` may encapsulate a device pointer when working with a CUDA backend, or a `cl_mem` with an OpenCL backend. Thus, SYCL objects could not be converted to native objects without knowing the backend they come from. To obtain the native representation of a SYCL object, a value of `sycl::backend`, which identifies a platform, need to be specified as a template parameter at compile time. Unwrapping a SYCL object also leads to a check at runtime. While the object being unwrapped comes from a backend different from the specified one, the program will fail (Fig. 1).

Though the specification defines abstractions and describes their behaviors, Khronos does not provide an official implementation of SYCL. Thus, vendors could bring up their own SYCL implementations. Popular implementations include OneAPI Unified Acceleration Fundation (2024) from Intel and AdaptiveCPP Alpay and Heuveline (2020). SYCL implementations differ from each other in details, but a SYCL program could still work well with them as long as both the program and the implementation comply with the standard.

2.2 MPI and vendor-specific library

MPI, abbreviation for Message Passing Interface, is the de facto industrial standard for communication and has been massively deployed in many domains. MPI defines a set of frequently used communication patterns and takes advantage of multi-process parallelism, meaning that multiple processes cooperate to work out the problem and exchange data using MPI. MPI can be built on top of high performance networks like Infiniband to further boost performance. Therefore, an optimized program using MPI can be easily extended to run on hundreds of processes on multiple nodes.

As computation is offloaded to accelerators like GPUs, MPI is also broadly employed to support communication among accelerators due to its simplicity and conciseness. However, it is unable to produce satisfying performance since MPI can only handle message sending and receiving on CPU, and thus messages have to be transferred between host and accelerators during communication. To ameliorate bottleneck of communication, accelerator vendors bring up communication libraries optimized for their accelerators. Vendor-specific libraries, which are also referred to as xCCL, outperform MPI by utilizing hardware communication interfaces in a more efficient way. Usually vendor-specific communication libraries achieve lower latency and higher throughput by enabling direct data transfers among devices bypassing the host, through universal communication channels like PCIe or special buses like NVLink on NVIDIA devices. At the same time, vendor-specific libraries are deeply dependent on particular devices and programming models and therefore are not general solutions to communication.

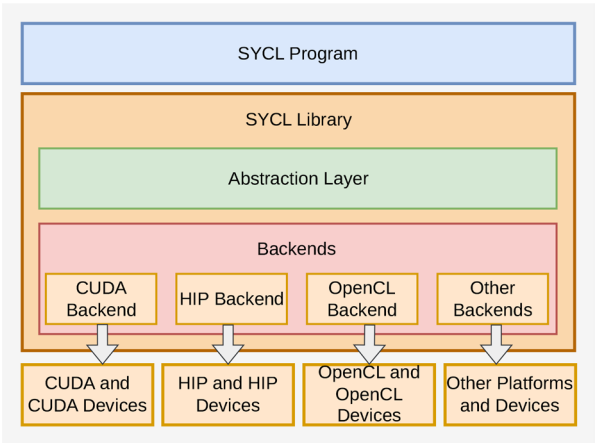


Fig. 1 Typical architecture of a SYCL implementation. The upper abstraction layer provides APIs and interacts with user programs, while backends implement SYCL abstractions and programming interfaces. Multiple backends could coexist in a SYCL installation. In such case, a backend will be selected automatically by the SYCL runtime or manually by the user to undertake compute task

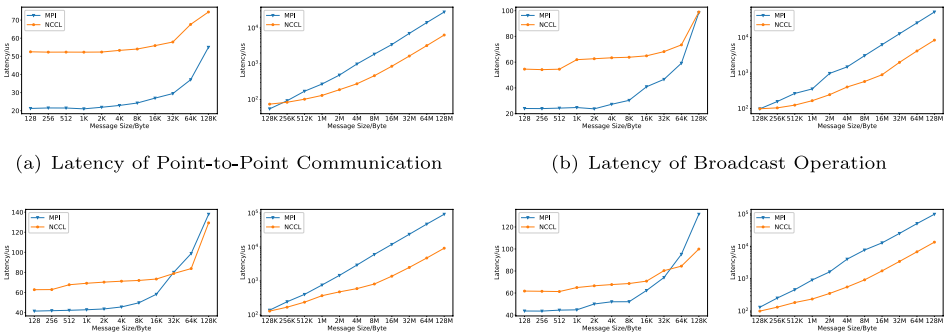
3 Challenges and motivation

While SYCL itself is only a specification of computing, SYCL programs resort to external communication libraries to benefit from the collaboration of multiple accelerators. Seeking for portable performance on different platforms, SYCL poses several requirements on co-working communication libraries:

1. Portability. While SYCL is ambitious to be a vendor-agnostic compute model and run on devices and platforms from any vendor, libraries cooperating with SYCL should not be limited to several particular platforms and devices, which hurts the portability of SYCL programs.
2. Performance. As accelerators are getting more and more powerful, they are more eager for data to remain busy. A high-performance communication library is critical to avoid communication from becoming the bottleneck and fully release the computational power of devices.
3. Openness. SYCL is designed to be open and is ready to accept newly emerging platforms and devices. Thus, it is more preferable that the library could catch up with SYCL and provide support for newly emerging platforms as soon as possible.

```
1 sycl::buffer<int, 1> buffer(bufer_size);
2
3 // other operations
4 // ...
5
6 {
7   sycl::host_accessor hacc{buffer};
8   MPI_Bcast(&hacc[0], buffer_size, MPI_INT, bcast_root, mpi_comm);
9 }
```

Listing 1 Example using MPI for communication in SYCL programs, usinig SYCL host_accessor feature



(a) Latency of Point-to-Point Communication

(b) Latency of Broadcast Operation

(c) Latency of Reduce Operation

(d) Latency of Allreduce Operation

Fig. 2 Performance of MPI and NCCL on a System with 4 NVIDIA A100 GPU

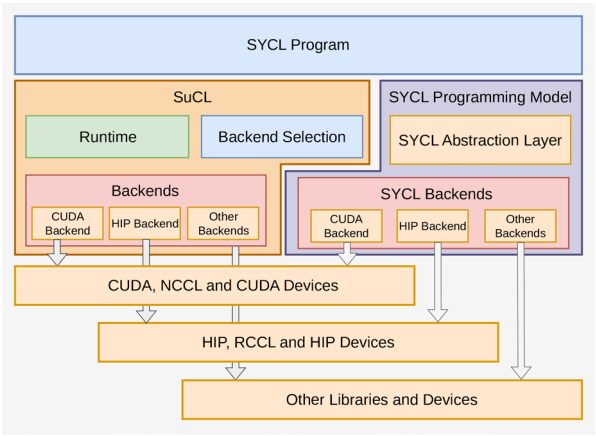


Fig. 3 Architecture of `SuCL` communication framework

For most libraries to work with SYCL, including MPI and vendor-specific libraries like NCCL, the major challenge derives from the dilemma of portability and performance. For MPI, a general solution to communication, performance yet remains a problem. Listing 1 shows an example using MPI within a SYCL program. As MPI routines accept pointers to message buffers as arguments, in line 7, a `host_accessor` is constructed with a SYCL buffer, and a legal pointer is then attained from the `host_accessor` and used by `MPI_Bcast` in line 8. Thus, MPI is platform-agnostic and portable solution for SYCL programs. However, the construction of `host_accessor` results in data movement from device memory to main memory where data reside until the destruction of `host_accessor`. Thus, with `host_accessors`, data have to traverse main memory during communication, incurring higher latency and degraded performance.

```
1  sycl::buffer<int, 1> buffer(buffer_size);
2
3  // other operations
4  // ...
5
6  queue.submit([&](sycl::handler& handler){
7    sycl::accessor acc{buffer, handler};
8    handle.host_task([=](const sycl::interop_handle& ihandle){
9      cudaStream_t stream = ihandle.get_native_queue<sycl::backend::ext_oneapi_cuda
       >();
10     void* ptr = (void*)ihandle.get_native_mem<sycl::backend::ext_oneapi_cuda>(acc);
11     ncclBcast(ptr, buffer_size, ncclInt, bcast_root, nccl_comm, stream);
12   });
13 });
14 queue.wait();
```

Listing 2 Example using NCCL for communication in SYCL programs, using SYCL native-interoperability feature

On the contrary, vendor-specific libraries, such as NCCL, prioritize performance over portability. By getting bound to particular devices and platforms, they are able to further optimize communication and boost performance by conducting direct data movement between devices, etc. Figure 2 shows results of several micro-benchmarks of MPI and NCCL on 4 NVIDIA A100 GPUs connected by PCIe, showing that NCCL outperforms MPI after message size reaches a particular threshold. On the other side, they are only available on particular platforms, and their utilization hurts the portability of a SYCL program. Listing 2 is an example using backend interoperability of SYCL and utilizing NCCL for communication in a SYCL program. To interact with native model, it is necessary to specify the platform and attain native objects from SYCL objects. In line 9 and line 10, the platform is specified by template parameter `sycl::backend::ext_oneapi_cuda`. The program is then limited to CUDA platform, and execution on any other platform will lead to a program failure.

To further improve performance, several MPI implementations are now GPU-aware, which means that they are also able to optimize data movement between accelerators or utilize corresponding vendor-providing libraries when supporting communication for particular devices. However, this could impose non-trivial burden on MPI maintainers, and may incur redundant effort supporting the same platform on different MPI implementations. After that, when working with SYCL, optimizations for particular platforms may face the same problem like vendor-specific libraries if they are platform-dependent.

As SYCL is gaining attention in areas like AI and HPC and is getting employed in data centers and super computers, the absence of a portable and high-performance communication library capable for SYCL greatly limits the application of SYCL. To solve the problem, another solution, apart from MPI and vendor-specific libraries, is required.

The key to the problem is the design capable to guarantee portability and performance at the same time, as the experience of MPI and vendor-specific libraries shows that sacrificing one for the other does not work well in SYCL. Thus, the new solution should overcome challenges that: 1) it should be able to work on any platforms(**C1**); 2)it should achieve high performance rival to that native libraries could provide(**C2**); 3) it is open so that little effort is needed to extend it to support new platforms(**C3**).

While (**C1**) and (**C2**) seem to contradict each other, the successful experience of SYCL has figured out a feasible means to achieve the goal. SYCL provides an abstraction layer and defines unified APIs on top of various platforms and programming models, with which SYCL developers are able to write codes ready to run on any platforms. Meanwhile, backends in SYCL interact with underlying programming models and provide well-defined implementations to ensure performance. New backends can also be integrated into a SYCL implementation to add support for new platforms. In summary, SYCL solves the problem by adding layers and introducing backends.

Similarly, a communication library could solve (**C1**) and (**C2**) through layered and backend-based architecture. Meanwhile, if backends can be easily added to the communication library, support for new platforms can also be quickly integrated into the library, thus solving (**C3**). However, neither MPI nor any of vendor-specific libraries adopts such design. Thus, we design a communication library and framework dedicated to SYCL, which adopts layered and backend-based architecture to work out the challenges.

4 Proposed design

We propose SuCL, a framework and library to support communication in SYCL. Prioritizing portability, SuCL serves as an abstraction layer upon various programming models and devices, bridging upper SYCL programs with underlying communication libraries. As is depicted in Fig. 3, SuCL adopts layered and backend-based design. Major components of SuCL include the runtime, backends and the backend-selection mechanism. The runtime interacts with upper SYCL programs, backends interact with underlying programming models as well as communication libraries, and the selection mechanism helps the runtime select the most proper backend for use.

For the sake of simplicity, SuCL adopts multi-process parallelism. SuCL programs are supposed to launch multiple processes, to each of which a SYCL device is attached in order to handle compute tasks. SuCL utilizes MPI to exchange control messages among host processes.

4.1 SuCL runtime

SuCL runtime sits on the top of SuCL framework and defines unified APIs for upper SYCL programs. Also, to work with most SYCL implementations, the runtime is fully compliant with SYCL 2020 specification, and avoids uses of any implementation-specific features. Thus, SYCL programs are supposed to work well with SuCL on any computing platform and SYCL implementations.

APIs defined by SuCL runtime can be divided into two categories: APIs to manage the framework itself and APIs to invoke communication operations. The former includes APIs to initialize and finalize the framework as well as those to query information about communication, and the latter consists of MPI-style communication APIs. Currently SuCL provides APIs to conduct communication in most frequently-used patterns, such as `send`/`recv` for point-to-point pattern, `reduce` for all-to-one pattern, `allreduce` for all-to-all pattern and `broadcast` for one-to-all pattern. Implementations of communication routines are left to backends.

SuCL runtime avoids getting coupled with any specific platforms. Thus, SuCL runtime interacts with SYCL programs and backends through SYCL objects. Along with unified APIs, this eliminates the need of exposing native representations in SYCL programs and inside the runtime during invocations of communication operations. Accepting SYCL objects also introduces additional problem: while SYCL programs are designed for various purposes, they use buffers with different element types and in different dimensions, adding difficulty to the design of SuCL communication APIs. We cope with the problem using SYCL buffer reinterpretion, which will be discussed later in 4.2.

While there are multiple devices available in the system, the one to submit kernels and participate in communication should be specified manually by the user and passed to the runtime during the initialization. The runtime then builds up a global view over all participating devices, recording necessary information such as rank and SYCL backend of each device. The view helps looking up a device during communication.

4.2 Backend and communication invocation

SuCL runtime does not cope with underlying libraries directly for the sake of portability and openness. Instead, SuCL introduces backends to interact with underlying computing platforms as well as communication libraries. Each backend is bound to one certain platform and is able to take advantage of corresponding communication libraries and platform-specific features. Through backends, SYCL programs utilize vendor-specific communication libraries indirectly.

Each communication API defined by the runtime owns an implementation in each backend. Backends guarantee performance by providing well-designed implementations of communication routines, and the correctness of the implementation is also ensured by the designer. Communication patterns can be implemented using wrapper functions which simply invoke corresponding library routines if such routines exist. In case that a library does not provide implementations for certain communication patterns, or alternative algorithms are more preferable for certain communication patterns, backend designers are allowed to build up their own implementations for those patterns using available routines provided by communication libraries. Consequently, there could be multiple backends, which differ in the way how a

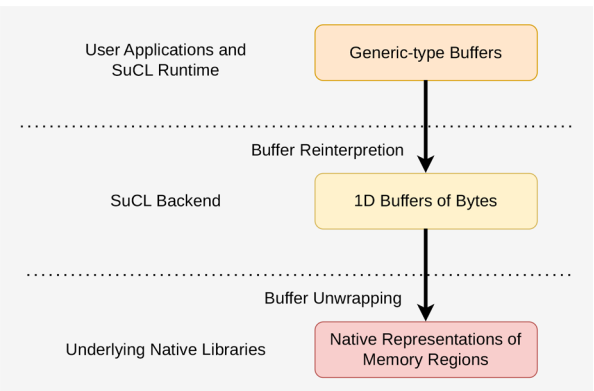


Fig. 4 Buffer transformation during communication invocations

particular communication operation is implemented, available for one platform.

On the invocation of communication operations, the runtime resorts to communication routines provided by selected backend, and finally correspondent native library APIs are invoked inside the backend. During the procedure, messages from user programs are passed from the runtime to underlying libraries. While the runtime accepts buffers in different shapes, backend routines accept 1D buffers in bytes only, and native libraries accept memory objects in other forms. Thus, messages are processed before they finally reach underlying communication libraries, as shown in Fig. 4. Inside SuCL runtime, messages provided by user programs and in different shapes are reinterpreted into 1D buffers in bytes using SYCL API, and passed to backend APIs. For `reduce` and `allreduce`, type of buffer element and reduction operation are passed to backends for later use. Then, inside the backend, buffers are further unwrapped to attain native memory representations using SYCL backend interoperability. Unwrapping inside backends does not incur runtime error, as selection mechanism ensures that backend is activated only when the program is running on the platform it targets.

To ensure a correct order of invocation, SuCL communication operations rely on SYCL runtime scheduling to synchronize with other compute kernels. In SYCL, constructions of an accessor in a command group create a dependency on particular buffer for submitted operation. For example, in line 7 in Listing 2, the construction of `acc` creates a dependency on `buffer` for the task submitted in line 8. With dependencies between different submissions, SYCL runtime is able to schedule tasks in a correct order. This ensures that communication operations will not start until dependent tasks finish. However, some native library routines, such as NCCL communication routines, are asynchronous, which means that communication operations may be marked as complete before the communication is truly complete. At the moment we do not have a satisfying solution to the problem and we simply place a synchronization after communication operations to avoid data race.

4.3 Backend selection

Backend selection is an important step performed by the runtime during the initialization of SuCL. The purpose of backend selection is to select and load the most suitable communication backend for devices involved in communication. Only after a proper backend is selected and initialized, initialization of the whole framework can be completed and communication can take place (Fig. 5).

Unlike in SYCL where backends can be detected and selected by SYCL runtime automatically, in SuCL, backend selection requires assistance from user using a configuration

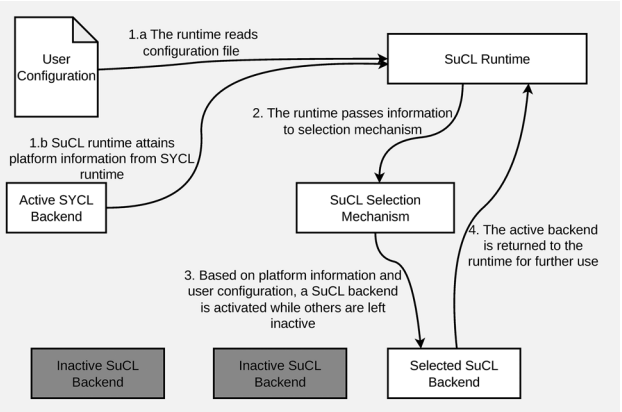


Fig. 5 Procedure of backend selection

file, because backend selection requires detailed information about platforms supported by a SYCL implementation. The SYCL specification leaves great freedom to SYCL implementations that, though the specification requires that SYCL backends are identified by `sycl::backends`, it does not specify the value for each platforms. Thus, different SYCL implementations may refer to the same platform through different values. To work with different SYCL implementations, SuCL requires users to inform the mapping between the value of `sycl::backend` and the platform before selecting SuCL backend. Another reason is that there may be multiple available backends targeting the same platform at the same time. While implementing a SYCL backend could be tough, implementing SuCL backends with tuned algorithms to get adjusted to particular use cases is quite straightforward. Thus, it is almost impossible for the runtime to foresee and detect every backend, and consequently, backends as well as their paths should be specified in the configuration file.

With the configuration file, SuCL runtime is now ready to select and initialize a backend for communication. The selection starts with querying platform information from the SYCL device specified by user and bound to the process. Then, the runtime checks whether a communication backend is appointed to the platform in the configuration file. If such a backend exists, the runtime loads and initializes the backend, otherwise, a default backend should be specified in the configuration file and is activated by the runtime. Since the default backend is used when no other backend is considered appropriate, it may be active when encountering a new computing platform with no support yet. Thus, to ensure normal functionality of the program, the default backend should be built on top of a portable and universal communication library, such as MPI.

Vendor-specific libraries do not remain the best solution all the time. In previous micro-benchmark evaluations on NVIDIA A100 and in previous works(Chen et al. 2023b;

Weingram et al. 2023), MPI still outperforms vendor-specific libraries in some cases when messages are small enough. Thus, a configurable threshold is added to SuCL so that messages with sizes under the threshold will travel through MPI, while those with sizes larger than the threshold travel through backends. The threshold could be attained from micro-benchmarks, and should be specified in the configuration file provided to SuCL. A threshold of 0 will force SuCL to use vendor-specific libraries all the time, and a threshold negative will disable vendor-specific libraries and use MPI for communication. Such design allows users to tune SuCL and enables SuCL to achieve optimal performance.

5 Example

In this section, we illustrate the basic usage of SuCL and the implementation of a backend using the following code samples.

5.1 Usage of SuCL

```
1  #include <sccl.hpp>
2
3  int main(int argc, char** argv){
4    sccl::scclInit(argc, argv);
5    sycl::device dev{sycl::gpu_selector_v};
6    sccl::Communicator* communicator = sccl::Communicator::newCommunicator(
       MPI_COMM_WORLD, dev);
7    sccl::Device* device = communicator->getLocalDevice();
8
9    sycl::buffer<int, 1> buffer(buffer_size);
10
11   device->getNativeQueue().submit([&](sycl::handler& handler){
12     ...
13   });
14
15   device->broadcast(buffer, {buffer_offset}, {buffer_size}, bcast_root);
16   device->wait();
17
18   ...
19
20   sccl::Communicator::destroyCommunicator(communicator);
21 }
```

Listing 3 Example of basic usage of SuCL.

Listing 3 shows a basic usage of SuCL. The SuCL runtime gets initialized in line 4 and line 6. In line 4, `sccl::scclInit` checks command line arguments, trying to recognize and load a user-specifying configuration file. MPI is also initialized in this step for later use. In line 6, a SYCL device is provided and bound to SuCL runtime. Then, the runtime builds up a global association between ranks and devices that is used in communication by exchanging necessary information using MPI. Also, with the given SYCL device, the runtime is then able to determine which communication backend to load and use. After that, SuCL is ready for use.

After initialization, both compute kernels and communication operations can be submitted through an `sccl::Device` handle. From line 11 to line 13, a kernel accessing the buffer is submitted to the device, and then in line 15, the device takes part in a broadcast communication. Finally in line 20, SuCL gets finalized and resources are released.

5.2 Implementation of a backend

```
struct Backend {
    sendFunc send = nullptr;
    recvFunc recv = nullptr;
    broadcastFunc broadcast = nullptr;
    reduceFunc reduce = nullptr;
    allreduceFunc allreduce = nullptr;
    ...
};
```

Listing 4 Definition of SuCL Backend

As is shown in Listing 4, backend in SuCL is a C++ structure holding function pointers to communication routines, therefore backend designers could extend the structure to accommodate extra information necessary and used by native libraries. For example, a CUDA backend may extends SuCL backend to hold a NCCL communicator required to invoke NCCL communication. Extra information is initialized during the initialization of the backend instance.

Listing 5 exhibits the implementation of broadcast operation in CUDA backend. In the example, the broadcast operation is finally mapped to an invocation of ncclBroadcast, in line 20 to line 22, using SYCL backend-interoperability feature. The address of the routine is then filled to the correspondent field of SuCL backend instance, during the creation and initialization of the instance.

```
1  sycl::event broadcast0(const Device &device, sycl::buffer<byte, 1> &buffer,
2                         const sycl::range<1> &offset, const sycl::range<1> &count,
3                         int root, const std::vector<sycl::event> &events){
4      const BackendCuda *backend =
5          (const BackendCuda*)device.getBackend();
6      int rootLocalRank = device.getDomain()->getDeviceLocalRank(root);
7      return device.getNativeQueue().submit(
8          [&](sycl::handler &handler)
9          {
10             handler.depends_on(events);
11             sycl::accessor acc{buffer, handler};
12             handler.host_task(
13                 [=](const sycl::interop_handle &ihandle)
14                 {
15                     byte *deviceptr = (byte*)ihandle.
16                         get_native_mem<sycl::backend::ext_oneapi_cuda>(acc)
17                             + offset.size();
18                     cudaStream_t nativeStream =
19                         ihandle.get_native_queue<sycl::backend::ext_oneapi_cuda>();
20                     ncclBroadcast(deviceptr, deviceptr, count.size(),
21                                   ncclInt8, rootLocalRank, backend->comm,
22                                   nativeStream);
23                 });
24         });
25 }
```

Listing 5 Implementation of broadcast operation in CUDA backend

6 Evaluation

6.1 Evaluation setup

Evaluations are conducted on two systems, NVIDIA system and AMD system. Detailed information of both systems is listed in Table 1. On the NVIDIA system, evaluation will make use of up to 2 nodes, each with 8 GPUs, and thus a maximum of 16 GPUs. Inter-node communication is supported by 200 G Infiniband network. `SuCL` would select MPI or NCCL for communication according to the threshold and message sizes during communication. On the AMD system, due to the lack of device, evaluation could only be conducted on 1 node with 2 devices. For communication libraries, MPI and RCCL are available on this system.

Evaluations are carried out with a series of micro-benchmarks and three applications. For micro-benchmarks, we implement OMB[7]-like micro-benchmarks in SYCL, and for application-level evaluation, three applications are ported to SYCL and used: Halo and Transpose from Zheng (2017) and CoMD from Compute Applications (2015).

6.2 Micro-benchmark evaluation

This section shows evaluation results of micro-benchmarks on NVIDIA system and AMD system. In micro-benchmarks we select `broadcast`, `allreduce` and `reduce` for examination. As for other communication patterns, at the time of writing they are still requiring more examinations in order to ensure the functionality and correctness, and are thus dismissed from the evaluation.

In micro-benchmarks, message sizes are increased from 128 Bytes to 128 MB. Before communication, a compute kernel is submitted to ensure that buffers reside in device memory, and synchronizations are inserted before and after communication. According to the test results on A100 system, we select 64 KB as the threshold switching MPI and native communication libraries (`SuCL`-t64k). We also test `SuCL` with the threshold of 0 (`SuCL`-t0). Figure 6 and 7 show results of micro-benchmark evaluation on NVIDIA system and AMD system respectively.

6.2.1 NVIDIA System

On NVIDIA system, the overall results for different communication patterns and configurations are similar. While NCCL yields a higher latency than MPI at the beginning, the latency of MPI finally overtakes that of NCCL, because the latency of NCCL grows much steadily than that of MPI as message size goes up gradually. As NCCL is able to utilize high-bandwidth NVLink for communication, NCCL ends up getting a latency dozens of times and even hundreds of times lower than that of MPI, while message sizes are large enough. On this system, `SuCL` introduces slight overhead, about several microseconds, thus `SuCL`-t0 basically mirrors NCCL in evaluations. For `SuCL`-t64K, it is able to yield the optimal latency most of time by switching to MPI when communication is handled by small messages and switching to NCCL to benefit from high-bandwidth interconnections as message sizes grow. However, results of most evaluations indicate that a threshold of 64KB may be a bit large for the system. When the message size is within the range of 16KB to 64KB, in most cases `SuCL`-t64K behaves worse than NCCL and `SuCL`-t0 due to a late switch to NCCL.

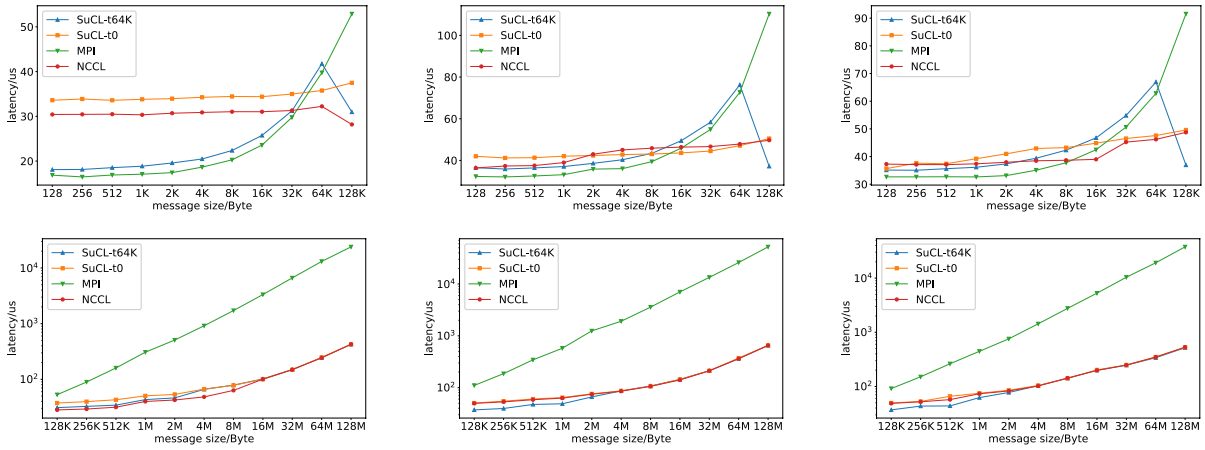
While more devices participate in communication, the proper value of threshold changes accordingly. Generally, a larger threshold is required as the number of participating devices increases. In the test of reduce, the proper threshold is 16KB when 4 GPUs take part in communication, and while the number of devices increases to 8 and 16, the proper value also increases to 32KB and 128KB. Similar phenomenon could be seen in the tests of broadcast and allreduce, but less evidently.

6.2.2 AMD system

Evaluation results on the AMD system show different cases. On AMD system, a notable overhead is seen on `SuCL`-t64K when `SuCL` is switching to MPI, making `SuCL`-t64K

Table 1 System Hardware and Software Information

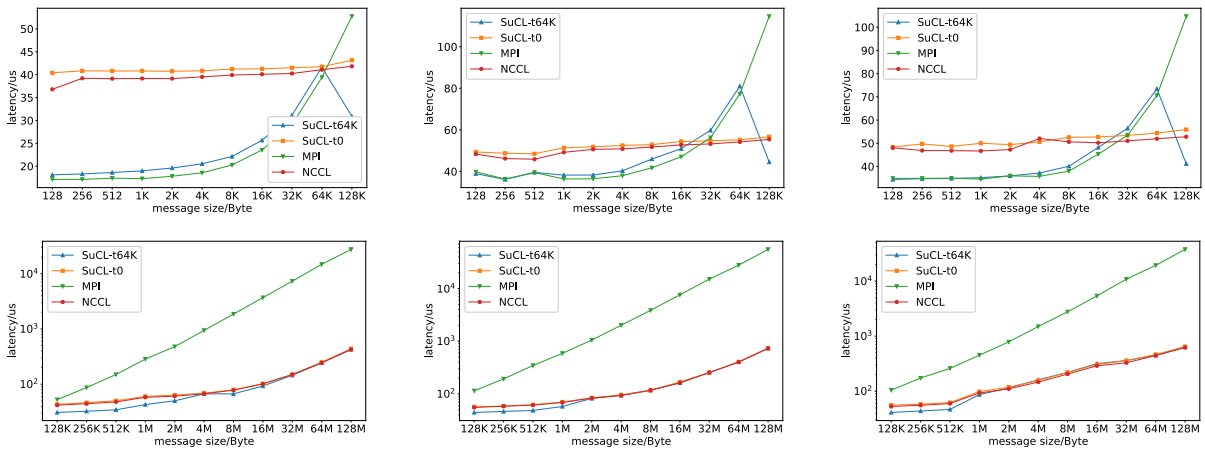
System Component	NVIDIA System	AMD System
CPU	Intel Xeon 8358P	AMD EPYC 7302
Memory	1T	256 G
Sockets	2	2
Cores/Socket	32	16
Computing Devices	8x NVIDIA H100 80 G with NVLink	2x AMD MI100 32 G
Computing Platform	CUDA 12.0.0	HIP 5.5.0
SYCL Implementation	Intel OneAPI 2024.2.1 with CUDA plugin	AdaptiveCPP 24.02 with Clang 17
MPI Implementation	Intel MPI 2021.13	MPICH 4.1.2
`SuCL` Backend	NCCL 2.19.3	RCCL 2.15.5



(a) Latency of Broadcast Communication on NVIDIA System (1 Node, 4 GPUs)

(b) Latency of Allreduce Communication on NVIDIA System (1 Node, 4 GPUs)

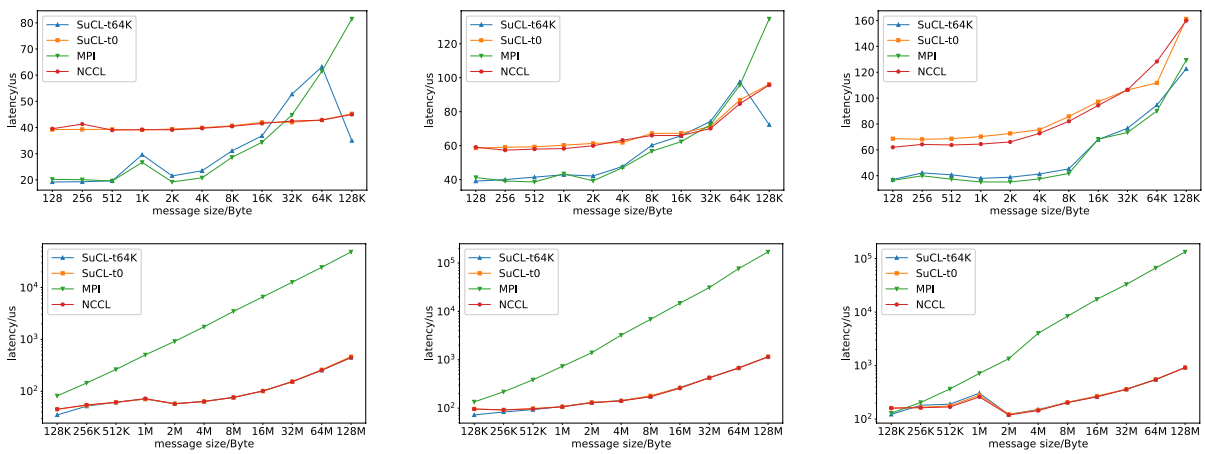
(c) Latency of Reduce Communication on NVIDIA System (1 Node, 4 GPUs)



(d) Latency of Broadcast Communication on NVIDIA System (1 Node, 8 GPUs)

(e) Latency of Allreduce Communication on NVIDIA System (1 Node, 8 GPUs)

(f) Latency of Reduce Communication on NVIDIA System (1 Node, 8 GPUs)

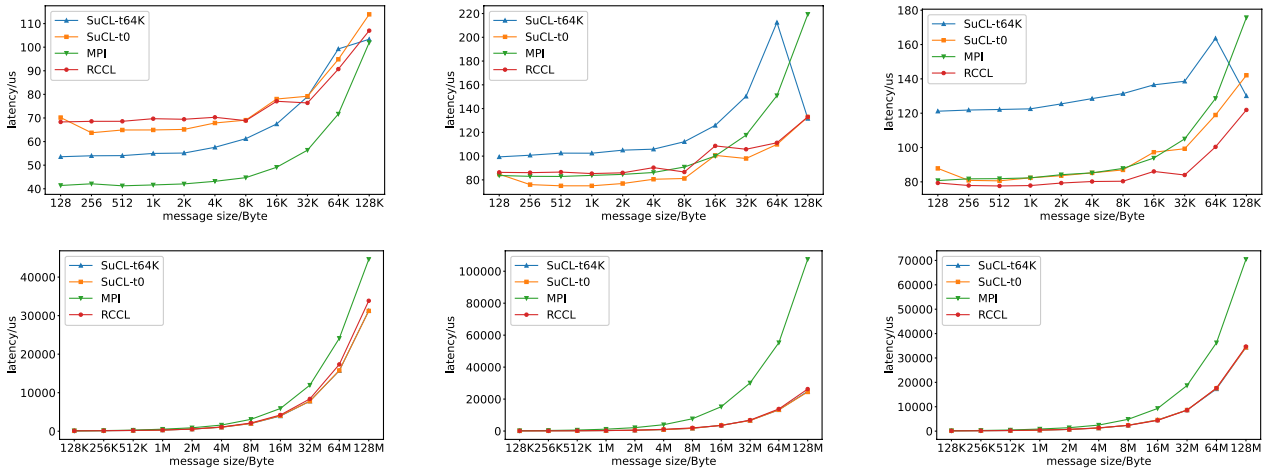


(g) Latency of Broadcast Communication on NVIDIA System (2 Nodes, 16 GPUs)

(h) Latency of Allreduce Communication on NVIDIA System (2 Nodes, 16 GPUs)

(i) Latency of Reduce Communication on NVIDIA System (2 Nodes, 16 GPUs)

Fig. 6 Result of microbench evaluation on NVIDIA System



(a) Latency of Broadcast Communication on AMD System (1 Node, 2 GPUs)

(b) Latency of Allreduce Communication on AMD System (1 Node, 2 GPUs)

(c) Latency of Reduce Communication on AMD System (1 Node, 2 GPUs)

Fig. 7 Result of microbench evaluation on AMD System

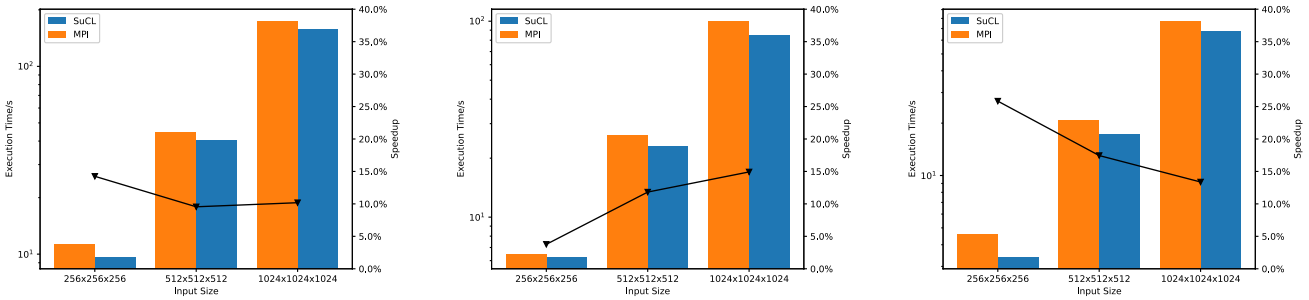
producing a suboptimal performance in the evaluation at small message size. The reason for that may be a naive compilation and the lack of optimization on host code.

For broadcast pattern, before message sizes reach 64KB, MPI is able to produce a lower latency. Due to the overhead, SuCL-t64K fails to mirror the performance of MPI, but still outperforms RCCL and SuCL-t0. After message sizes exceed 64KB, latency of MPI starts to overtake that of RCCL, and increases faster. Since both SuCL-t0 and SuCL-t64K utilize RCCL at this moment, they also yield latencies evidently lower than that of MPI, and nearly mirror the performance of RCCL. Finally, SuCL-t64K and SuCL-t0 gain a speedup of 30% when the message size reachs 128 M.

For reduce and allreduce, however, the case is distinct. The major difference sits on the performance at small message sizes. Before message sizes reach 64KB, RCCL yields latency similar to that of MPI. Thus, due to the overhead, SuCL-t64K produces the worst latency, and the performance of SuCL-t0 is more closer to the optimal one. After message sizes exceed 64KB, both SuCL-t64k and SuCL-t0 take advantage of RCCL and are able to outperform MPI. SuCL-t64k and SuCL-t0 end up gaining speedups more than 70% over MPI in allreduce when message size reaches 128MB, and about 50% in reduce.

6.3 Application evaluation

In this section we evaluate SuCL and compare it with MPI through three applications: Halo, Transpose and CoMD. For all of the applications, they are configured to run 100 iterations. For each iteration in Halo, a compute kernel is submitted, and before and after communication, two kernels for packing and unpacking messages respectively are submitted. For Transpose, no compute kernel is submitted, but

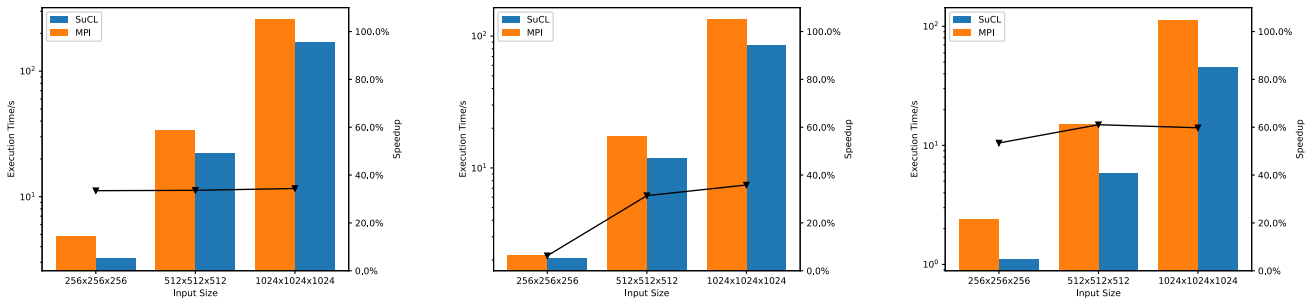


(a) 1 Node, 4 GPUs

(b) 1 Node, 8 GPUs

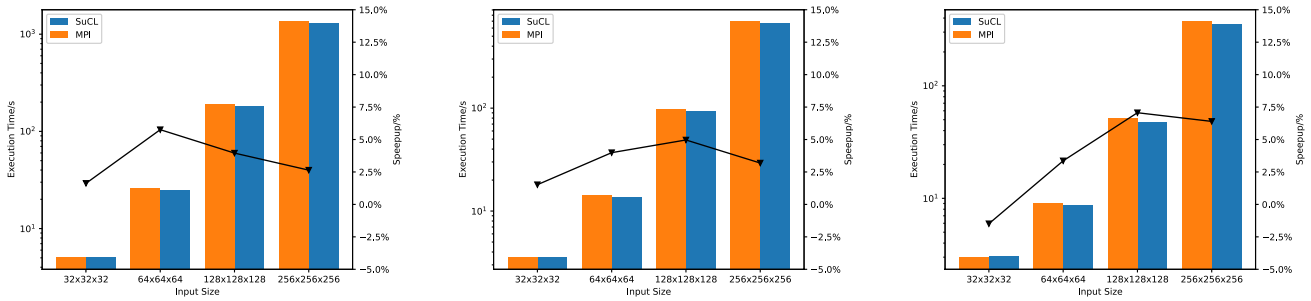
(c) 2 Node, 16 GPUs

Fig. 8 Runtime and speedup of Halo on Nvidia system. The fold line indicates the speedup of SuCL over MPI



(a) 1 Node, 4 GPUs (b) 1 Node, 8 GPUs (c) 2 Node, 16 GPUs

Fig. 9 Runtime and speedup of Transpose on Nvidia system. The fold line indicates the speedup of SuCL over MPI



(a) 1 Node, 4 GPUs (b) 1 Node, 8 GPUs (c) 2 Node, 16 GPUs

Fig. 10 Runtime and speedup of CoMD on Nvidia system. The fold line indicates the speedup of SuCL over MPI

kernels for packing and unpacking remain, thus Transpose is less compute-intensive than Halo. CoMD, on the other hand, submits a batch of compute kernels in an iteration, and is much more compute-intensive than Halo.

In application-level evaluation, we continue to set a threshold of 64KB for SuCL to make an accordance with micro-benchmarks.

6.3.1 NVIDIA system

Figure 8 shows the result of Halo on NVIDIA system. SuCL gains a minimum speedup of about 4% in the evaluation when the input size is $256 \times 256 \times 256$ and running on 8 GPUs. In other cases, SuCL outperforms MPI by more than 10%, and the maximum speedup is about 25% with an input size of $256 \times 256 \times 256$ when running on 2 nodes, 16 GPUs.

Figure 9 shows the result of Transpose on NVIDIA system. As Transpose is more communication-intensive than Halo, SuCL achieves higher speedups in Transpose than in Halo. The same as in Halo, SuCL gains the minimum speedup with an input size of $256 \times 256 \times 256$ on 8 GPUs, about 6% faster than MPI. Apart from this, when running on 4 GPUs or 8 GPUs on only 1 node, SuCL runs faster than MPI by more than 30%, and when running on 16 GPUs on 2 nodes, SuCL outperforms MPI by about 60%. SuCL achieves the maximum speedup of 61% at the input size of $512 \times 512 \times 512$ when running on 16 GPUs.

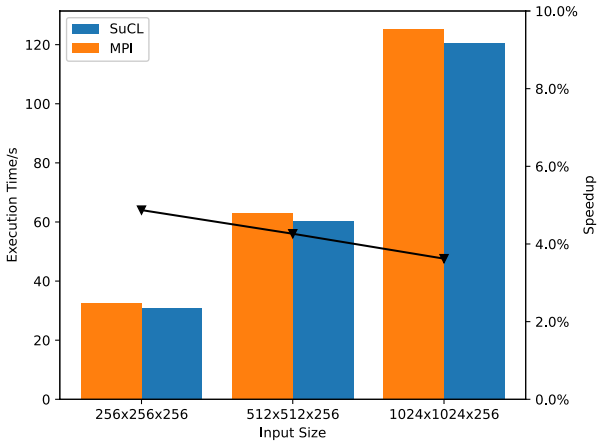


Fig. 11 Runtime and speedup of Halo on AMD system (1 Node, 2 GPUs). The fold line indicates the speedup of SuCL over MPI

Figure 10 shows the evaluation results of CoMD on NVIDIA system. Unlike Halo and Transpose, CoMD is much more compute-intensive. In CoMD, the maximum speedup is only 7%, achieved at $128 \times 128 \times 128$ and running 16 GPUs. At $32 \times 32 \times 32$ on 16 GPUs, `SuCL` is slightly slower than MPI. In other cases, the speedups vary from 1.5 to 6%.

6.3.2 AMD system

Unfortunately, both versions of CoMD(MPI version and `SuCL` version) fail to execute normally on AMD system, mainly due to a wrong configuration of the system hardware or the system software. Thus, we dismiss the result of CoMD on AMD system.

Figure 11 demonstrates result of Halo on AMD platform. `SuCL` runs slightly faster than MPI, gaining a maximum improvement of about 5% when input size is $256 \times 256 \times 256$. Improvements for input sizes of $512 \times 512 \times 256$ and $1024 \times 1024 \times 256$ are around 4%. One possible reason for the poor improvement is that too few devices take apart in the computation. While the number of devices reduces by half, computation for each device is doubled but the amount of data exchange remains unchanged. Thus, computation makes up the major part of execution time, and the improvement on communication is hardly reflected on the overall runtime.

Figure 12 displays the result of Transpose on AMD platform. `SuCL` still runs notably faster than MPI and reduces 30% execution time for all input sizes. This may be due to the absence of compute kernel in Transpose, and thus the total run time is dominated by communication, which is hardly affected by the reduction of devices.

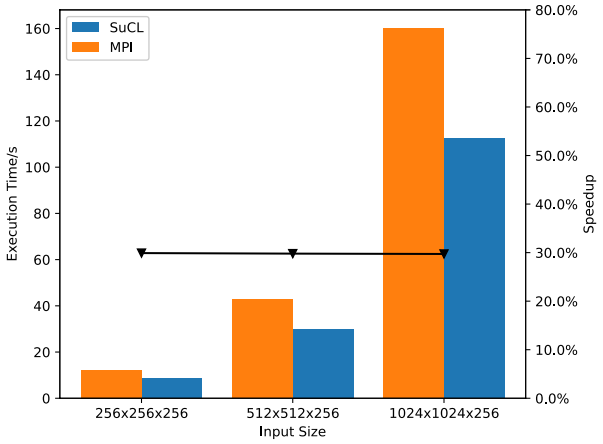


Fig. 12 Runtime and speedup of Transpose on AMD system (1 Node, 2 GPUs). The fold line indicates the speedup of SuCL over MPI

7 Related work

Up to now, major accelerator vendors have brought up their communication libraries supporting their devices. Nvidia develops NCCL Nvidia (2024) to support CUDA devices, and AMD and Intel propose RCCL AMD (2025) and oneCCL [12] respectively. These libraries could utilize special hardware communication interfaces to provide higher bandwidth and lower latency, but they are deeply dependent on platforms and support limited ranges of devices.

Wang et al. (2014) explore the way to integrate CUDA support into MPI. They optimize data transfers for both contiguous data and non-contiguous data, and pipeline data transfers between GPU memory and host buffer. They also utilize unified addressing provided by CUDA to ease programming difficulty. Shafie Khorassani et al. (2021) shares their experience on add support and optimization for AMD GPU to MPI. As ROCm exposes similar APIs to CUDA, they design a unified API layer to avoid duplicated effort. They optimize data transfers using features like ROCmRDMA, mapped memory copy and ROCm IPC. Chen et al. (2023a) extend MPI to support newly emerging Intel GPUs using prior techniques used to support Nvidia and AMD devices, such as CPU staging and pipelining. Though these works extend MPI to support GPUs from various vendors, they do not utilize communication libraries from vendors and specific communication hardware to further boost performance. Also, these works exhibit little openness to further support new devices.

Weingram et al. (2023) demonstrates communication patterns, algorithms and popular collective communication libraries for deep learning, including MPI and NCCL. They also conduct evaluations on communication libraries, which show that NCCL and MSCCL, a collective communication library based on NCCL, outperform MPICH, a CUDA-aware MPI implementation.

Chen et al. (2023b) propose MPI-xCCL, which extend GPU-aware MPI to make use of vendor-specific communication libraries. They propose an abstraction layer as well as a set of communication APIs, and upper MPI-xCCL APIs are mapped to underlying APIs from vendor-specific libraries. MPI-xCCL also adopts hybrid design that switches between MPI and vendor-specific libraries to gain best performance. However, their design does not adopt backend-based architecture and lacks selection mechanism, thus is less configurable and flexible.

8 Conclusion and future work

In this article, we propose SuCL, an open communication library and framework cooperating with SYCL. By adopting layered and backend-based design, SuCL guarantees portability and performance at the same time. SuCL provides unified communication APIs for SYCL programs and relies on well-implemented backends, which can be added to the framework easily, to provide support for various platforms. In micro-benchmarks, SuCL could achieve optimal performance most of time by switching between MPI and vendor-specific libraries, and significantly outperforms MPI as message sizes grow. The result also shows that a single threshold may not be capable in all cases. In application evaluations, SuCL works better on communication-intensive applications than on computation-intensive ones, and achieves up to faster than MPI.

Future work of SuCL includes adopting more flexible threshold strategy, adding new backends to support more platforms, integrating a communication kernel generator to help implementing backends, enabling SuCL to utilize devices from different vendors at the same time and implementing load-balance mechanism.

Author Contributions Xianwei Zhang managed this project. Hengzhong Liang conceived and implemented the framework. Han Huang helped in overall design and implementation. Hengzhong Liang, Han Huang and Xianwei Zhang analyzed the data. All authors read and approved the manuscript.

Funding This research is supported by the National Key R&D Program of China (Grant No. 2023YFB3002202).

Data availability Code and original data available on request from the authors.

Declarations

Conflict of interest The authors declare that they have no competing interests.

Ethics approval and consent to participate Not applicable.

References

Alpay, A., Heuveline, V.: Sycl beyond opencl: the architecture, current state and future direction of hipsycl. In: Proceedings of the International Workshop on OpenCL. IWOCL '20. Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3388333.3388658

AMD: ROCm communication collectives library (RCCL). https://github.com/rocm/rccl

Cai, Z., Liu, Z., Maleki, S., Musuvathi, M., Mytkowicz, T., Nelson, J., Saarikivi, O.: Synthesizing optimal collective algorithms. In: Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP '21, pp. 62–75. Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3437801.3441620

Chen, C.-C., Khorassani, K.S., Kuncham, G.K.R., Vaidya, R., Abduljabbar, M., Shafi, A., Subramoni, H., Panda, D.K.: Implementing and optimizing a gpu-aware mpi library for intel gpus: early experiences. In: 2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid), pp. 131–140 (2023). https://doi.org/10.1109/CCGrid57682.2023.00022

Chen, C.-C., Shafie Khorassani, K., Kousha, P., Zhou, Q., Yao, J., Subramoni, H., Panda, D.K.: Mpi-xccl: A portable mpi library over collective communication libraries for various accelerators. In: Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis. SC-W '23, pp. 847–854. Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3624062.3624153

Compute applications. https://github.com/AMDComputeLibraries/ComputeApps

Intel: Intel® oneAPI Collective Communications Library. https://oneapi-src.github.io/oneCCL

Intel: Intel MPI Library. https://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library.html

MPICH | High-Performance Portable MPI. https://www.mpich.org/

MVAPICH : Benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/

November 2024 TOP500. https://www.top500.org/lists/top500/2024/11/

Nvidia: NVIDIA collective communications library (NCCL). https://developer.nvidia.com/nccl

oneAPI Programming Model. Unified acceleration fundation. https://www.oneapi.io/

Portability of oneCCL. https://github.com/oneapi-src/oneCCL/issues/98

Shafie Khorassani, K., Hashmi, J., Chu, C.-H., Chen, C.-C., Subramoni, H., Panda, D.K.: Designing a rocm-aware mpi library for amd gpus: Early experiences. In: Chamberlain, B.L., Varbanescu, A.-L., Ltaief, H., Luszczek, P. (eds.) High Performance Computing, pp. 118–136. Springer, Cham (2021)

Shah, A., Chidambaram, V., Cowan, M., Maleki, S., Musuvathi, M., Mytkowicz, T., Nelson, J., Saarikivi, O., Singh, R.: TACCL: Guiding collective algorithm synthesis using communication sketches. In: 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23), pp. 593–612. USENIX Association, Boston, MA (2023). https://www.usenix.org/conference/nsdi23/presentation/shah

SYCL 2020 Specification (revision 8). The Khronos SYCL Working Group. https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html

SYCL Overview. The Khronos Group Inc. https://www.khronos.org/sycl/

Wang, H., Potluri, S., Bureddy, D., Rosales, C., Panda, D.K.: Gpu-aware mpi on rdma-enabled clusters: design, implementation and evaluation. IEEE Trans. Parallel Distrib. Syst. **25**(10), 2595–2605 (2014). https://doi.org/10.1109/TPDS.2013.222

Weingram, A., Li, Y., Qi, H., Ng, D., Dai, L., Lu, X.: xccl: a survey of industry-led collective communication libraries for deep learning. J. Comput. Sci. Technol. **38**(1), 166–195 (2023). https://doi.org/10.1007/s11390-023-2894-6

Zheng, Y.: Performance and scalability of communications in atmospheric model for exascale supercomputer. https://doi.org/10.5281/zenodo.1066934