Simulation of NCCL with SimGrid

Jean Conan
TU Wien

Technical Report

January 10, 2025

January 10, 2025

Abstract

This article demonstrates how GPU-to-GPU communications can be modeled using SimGrid by implementing a subset of the CUDA Runtime Library and NCCL with SimGrid's high-level interface (s4u).

Keywords: GPU communication, SimGrid, CUDA Runtime, NCCL, Simulation

1 Introduction

GPUs play a crucial role in modern high-performance computing (HPC) systems, and efficient communication between them is critical for performance. The NCCL (NVIDIA Collective Communications Library) is widely used for optimized multi-GPU communications. In this report, we demonstrate how GPU-to-GPU communications can be modeled using SimGrid [1], a simulation framework, by implementing a subset of the CUDA Runtime Library [2] and NCCL [3].

The objective of this work is to assess the feasibility of SimGrid in modeling NCCL and CUDA features for GPU communication. We present the methodology, implementation details, and preliminary results based on simulations of real-world communication patterns.

Glossary

Activity In SimGrid, an activity is an elemental action of the system. Example: a communication between two Hosts, a computation on one Host, a write on a Disk.... 1, 2

Actor A SimGrid actor represents the process to simulate. It can interact with the system by launching activities, suspending actors, killing actors, shutting down Hosts.... 1, 2

Host In SimGrid, represents a resource of computational power (e.g., CPU, GPU, one CPU core...). 1, 2

2 Methodology

In this section, we describe the methods used to simulate GPU communication using SimGrid, covering both the theoretical and practical aspects. We detail the modeling of GPUs, the integration of CUDA Runtime features, and the extension of SimGrid to support NCCL communication primitives.

2.1 Modeling a General-Purpose GPU

2.1.1 SimGrid Philosophy

SimGrid uses a resource-based logic to describe hardware. CPUs are typically modeled by computational power resources (s4u Hosts), and interconnection links are modeled by bandwidth resources (s4u Links).

It is logical to model a GPU as a Host, as it is also a computational resource. What differentiates a CPU from a GPU is how they interact with resources. A simulated CPU algorithm (s4u Actor) launches Activitys on the host it is located on, while a simulated GPU algorithm is run on a CPU, which then launches Activitys on the associated GPUs.

2.1.2 A Stream-Controlled Host

Using SimGrid's extension mechanism, a queue of "kernel calls" is linked to a simple Actor called the "Stream Actor". The Stream Actor behaves as follows:

```
while true do
  if stream is empty then
    suspend
  else
    kernel = stream.pop()
    consume(kernel)
  end if
end while
```

When a new kernel call is added by the CPU Actor that created the stream, it resumes the stream Actor, which then consumes all kernel calls. Consuming a kernel call means translating it into Activitys on the GPU and waiting for these Activitys to complete.

2.2 CUDA Runtime Library

NCCL-tests (a collection of tests for NCCL) use a small portion of the CUDA Runtime Library, primarily stream management and synchronization. The most recent version also uses CUDA Graphs.

2.2.1 Stream Management and Synchronization

The most important functions to cover are:

- `cudaSetDevice(int device)`
- `cudaStreamCreateWithFlags(cudaStream_t *pStream, unsigned int flags)`
- `cudaStreamSynchronize(void)`

The first two functions create a Stream object associated with a GPU. In the `cudaStreamCreateWithFlags` call, there is no option to specify which GPU Host to use, so a SimGrid extension is used to store the GPU Hosts accessible by the CPU. The `cudaStreamSynchronize` function allows the CPU to wait for all kernels to finish on the GPU. In the implementation, the CPU probes the stream actor to check if it is suspended, and if it isn't, it suspends itself.

Other CUDA Runtime functions used by NCCL-tests include `cudaMalloc`, `cudaHostMalloc`, `cudaHostFree`, `cudaFree`, and `cudaMemcpy`.

2.2.2 CUDA Graphs

In the most recent NCCL-tests, CUDA Graphs are utilized. A CUDA Graph is a set of kernels that can be launched together and multiple times. This mechanism has advantages such as easier communication of kernel calls to the GPU and simplifying the repeated launch of the same kernel.

There are two main objects: `CudaGraph` and `CudaGraphExec`. A `CudaGraph` can be modified, while a `CudaGraphExec` can be launched. In NCCL-tests, CUDA Graphs are created using the capture method: first, `cudaStreamBeginCapture` is called, then all the kernel calls are

issued, followed by `cudaStreamEndCapture`. Afterward, the `CudaGraph` is converted into a `CudaGraphExec` using `cudaGraphInstantiate`, and it is launched with `cudaGraphLaunch`.

To implement this, a high-level stream object was created. If there is no capture, it passes activities to the lower-level stream; if in capture mode, it adds the activities to the graph.

Since CUDA Graphs are essentially graphs of kernels, and kernels are graphs of activities in our simulation, CUDA Graphs can be represented as graphs of activities. However, CUDA Graphs can be launched multiple times, and kernels may involve data-dependent operations, requiring activities to be computed in the simulation as well.

To address this, a structure called `GpuActivity` was created. Alternatively, an extension of s4u Tasks could have been used.

2.3 NCCL

2.3.1 Communicators and Basic Communication

NCCL communicators are objects used for every communication. They are created from a unique ID. In NCCL, the ID is a constant-length string, but in the simulated library, an integer is used for simplicity. To create a communicator with multiple CPUs, one CPU generates the unique ID and broadcasts it to the others.

In the simulated implementation, `ncclComm` points to an object that holds a mapping of ranks to GPUs.

2.3.2 Collectives

NCCL collectives have been implemented as kernels, using `ncclGroup`, `ncclSend`, and `ncclRecv`. NVIDIA's documentation on communication patterns was partial, so additional information was obtained from third-party sources.

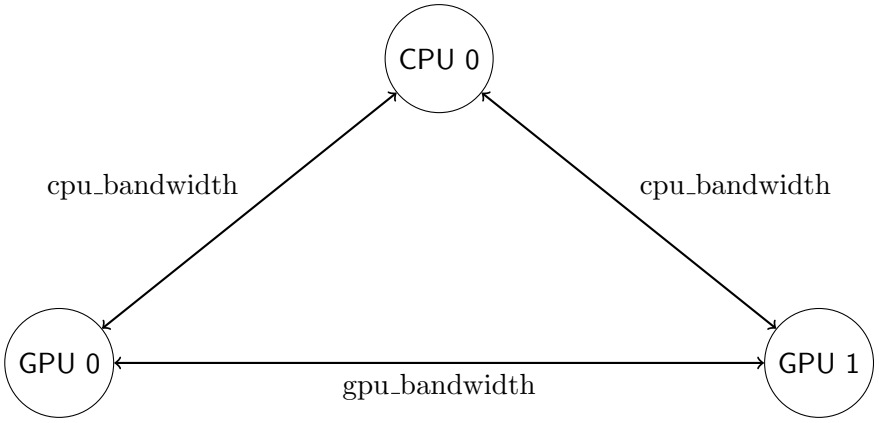
2.4 SMPI

The simulation library can be used with SMPI, for example, to communicate `ncclUniqueId` to create a common communicator with multiple CPUs. One issue that may arise is deadlock detection: if all processes involved in the SMPI application get suspended, an SMPI internal mechanism triggers a deadlock error, even if there isn't one (e.g., when all CPUs are waiting for corresponding GPUs for synchronization). To avoid this, a blank SMPI process can be added to wait on a final barrier before calling `MPI_Finalize` and exiting.

3 Experiments

NCCL-tests were run on Tesla GPUs, and the AMD equivalent RCCL-tests were run on MI100 GPUs on the Chameleon Cloud. Only one node and two GPUs were used for each test, so comparison at scale between reality and simulation has not been evaluated.

A simple model for a single node with two GPUs is illustrated as follows:



The measured bandwidth for NCCL communications corresponds to the GPU bandwidth. For communications other than AllReduce, Reduce, and ReduceScatter, the computational power of the GPU plays no role, and the CPU-to-GPU bandwidth only matters for very small communication sizes (comparable to kernel call sizes).

4 Conclusion

In this report, we have demonstrated that SimGrid can successfully model GPU-to-GPU communications using NCCL and the CUDA Runtime Library. While the experiments presented here focus on a small-scale simulation, the approach can be extended to larger systems. The ability to model detailed communication and computational behavior of GPUs in SimGrid opens up possibilities for performance prediction and optimization in HPC systems.

4.1 Future Work

Future work could include:

- Extending the model to support Hip Runtime (for AMD GPU).
- Add in place collectives support
- Adding a snvcc compiler to function like `smpi_cc` for `mpi_cc` : compiling native code directly into SimGrid.
- Evaluating the performance of other NCCL collectives at scale.
- Incorporating more detailed performance metrics for comparison with real-world results.

References

[1] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. Versatile, scalable, and accurate simulation of distributed applications and platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917, June 2014.

[2] NVIDIA. Cuda runtime library. Online Resource. Documentation available online at: https://docs.nvidia.com/cuda/cuda-runtime-api.

[3] NVIDIA. Nvidia collective communication library (nccl). Online Resource. Documentation available online at: https://docs.nvidia.com/deeplearning/nccl.