

Reducing Confusion and Broadcasting Understanding: An Empirical Study of the Semantics of NCCL

Max Stuppel, Alastair F. Donaldson, Azalea Raad
Department of Computing, Imperial College London, UK
{max.stuppel21, alastair.donaldson, azalea.raad}@imperial.ac.uk

Abstract—We present a foundation for formalising the semantics of NCCL, the NVIDIA Collective Communications Library. NCCL is a key component in distributed machine learning and HPC stacks that utilise NVIDIA GPUs. By combining analysis of the official documentation with empirical testing of the library, we present several key insights that characterise the concurrent programming model of NCCL. This includes the introduction of the concept of command matching for NCCL, and novel explanations of the conditions under which undefined behaviour may be observed.

Index Terms—NCCL, semantics, GPU concurrency

I. INTRODUCTION

The NVIDIA Collective Communications Library (NCCL, pronounced “nickel”) is a vendor-provided library for the CUDA programming model [1], providing high-bandwidth, topology-aware implementations of common multi-GPU routines such as all-reduce and all-gather [2]. These routines are highly optimised for data transfers over PCIe, RDMA, or NVIDIA’s own NVLink/NVSwitch, making them ideal for use in AI and HPC compute clusters [3], [4], [5]. Furthermore, due to NVIDIA’s dominant position in the AI accelerator market, with an estimated market share of over 80% [6], NCCL is widely used for distributed AI/ML workloads. It is the preferred backend for most major ML libraries including PyTorch, TensorFlow, and JAX [7], [8], [9].

Despite its widespread use, NCCL has relatively brief documentation [2] and no formal semantics. Crucial parts of the library are not documented rigorously, such as the semantics of command groups [10], and the difference between single-threaded and multi-threaded patterns for issuing operations. Given that many higher-level tools depend on NCCL, there is a clear need to bridge these gaps in documentation in order to ensure that machine learning and GPU-accelerated HPC stacks are robust and reliable.

To take a rigorous approach to finding bugs in NCCL, and ensure the correctness of its implementations — which cover a broad range of possible hardware and network configurations — it is a prerequisite to lay down a full formal semantics to describe the library’s intended behaviour. As a first step to that end, we present the results of a practical investigation into the semantics of NCCL, based on a thorough review of the available documentation and an empirical study of the library’s behaviour on real hardware. We have developed a

litmus testing framework to express test cases concisely and clearly, and facilitate running experiments reproducibly and at scale on real hardware. We provide a selection of examples that illuminate key behaviours of NCCL, and which may be non-obvious or unintuitive. Finally, backed by our test results, we provide novel, intuitive descriptions of common NCCL programming patterns and concepts, including concurrency, command groups, and command matching.

Data availability. We make our litmus testing framework and test suite available as open source at <https://github.com/maxstu/nccl-empirical-semantics>, with an archive of the exact code used in this paper via Zenodo [11].

II. UNDERSTANDING NCCL

We present the technical background on CUDA and NCCL, followed by a deep dive on several important NCCL concepts, with insights gleaned from the library documentation and through discussion with a NCCL expert from NVIDIA.

CUDA and NCCL. To understand NCCL as a library, we should first review the basics of CUDA [1], the language it targets. CUDA is a programming language based on C++ with the additional ability to define functions that run on external CUDA devices (typically GPUs), called *kernels*, and dispatch these kernels to run on available devices.

As GPU execution of kernels is asynchronous with execution on the host, it may be necessary for the programmer to impose explicit orderings or synchronisation points between them. There are many tools available in the CUDA ecosystem that target this problem [12], [13], [14], but the one primarily used with NCCL is CUDA *Streams* [15]. Streams are ordered first-in-first-out (FIFO) queues of work to be executed by a particular device, and there may be many streams on each device. CUDA provides the guarantee that, given two kernels issued towards the same stream, the kernel issued first will be completed before execution of the second kernel begins.

In addition to ensuring kernels are executed in the correct order on the device side, streams can be used to synchronise the host with the device. Calling `cudaStreamSynchronize(stream)` will cause the host thread to block until all operations on `stream` are complete [16]. Similarly, the programmer can invoke `cudaDeviceSynchronize` to block the host thread until operations on the current device are complete [17].

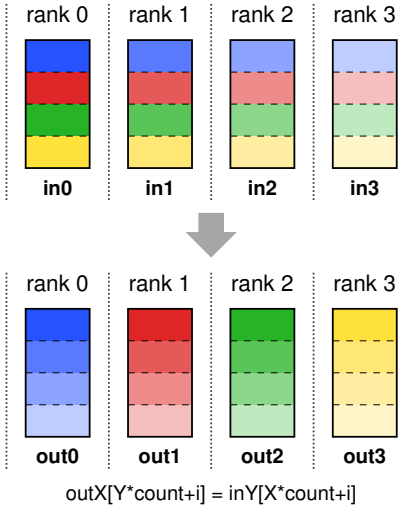


Fig. 1. All-to-all collective

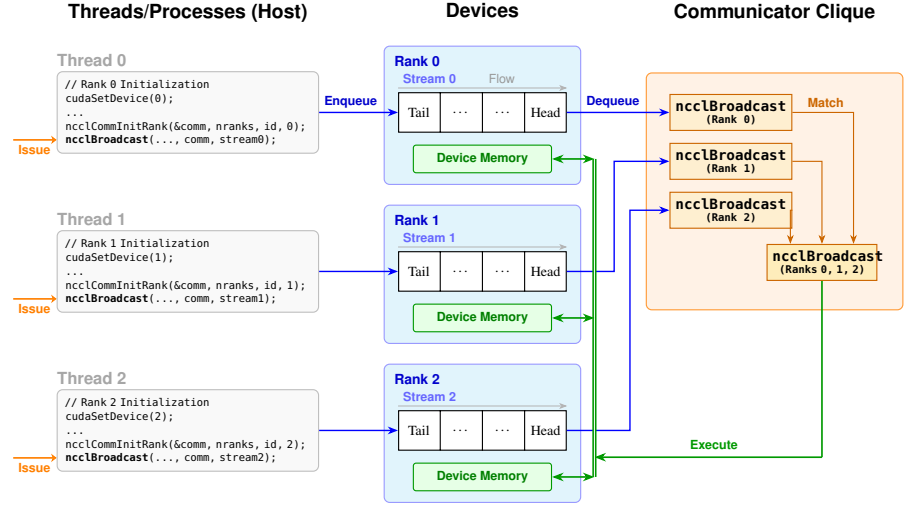


Fig. 2. Illustration of command issue, matching, and execution

As many devices may be attached to a single host, CUDA allows the programmer to designate which device to target by calling `cudaSetDevice(device)`. Doing so will cause all subsequent kernel launches by the same thread to target device until `cudaSetDevice` is called again to set another target. This mechanism can be used to manage multiple GPUs from a single thread sequentially, or, by spawning multiple threads, one can manage several GPUs in parallel by assigning each thread to manage a distinct device.

However, if the programmer wants to work with multiple GPUs concurrently, it can be very challenging to synchronise between all threads and devices involved. This is due to the need to strike a fine balance that ensures synchronisation is sufficient for correctness, but which avoids any unnecessary stalls where one thread or device waits on another.

NCCL facilitates managing multiple GPUs connected within a node or across a cluster in a uniform fashion. It provides routines for common communication patterns that are optimised for various network topologies, offering extremely low latency and high throughput [18].

Collective Operations. The most important of these routines are *collective operations*. These encompass eight operations critical for distributed machine-learning workloads: broadcast, scatter, gather, reduce, all-gather, all-reduce, reduce-scatter, and all-to-all [19]. Each collective necessarily operates not only on a single value, but on an entire buffer of data, which may be modelled as an array. For example, `ncclAlltoAll`, illustrated in Fig. 1, exchanges the i^{th} segment of the buffer on the j^{th} rank with the j^{th} segment from the i^{th} rank — in effect performing a matrix transposition across ranks.

NCCL provides highly optimised implementations of each collective for various common network topologies, which is critical to avoid bottlenecks. For example, distributed gradient descent requires an all-reduce at each step in order to

keep weights synchronised across nodes. A poorly optimised implementation can introduce excessive synchronisation stalls, which drastically lowers cluster utilisation as the process becomes network-bound rather than compute-bound [18].

Communicators and Point-to-Point Operations. To determine which collection of GPUs should participate in a particular collective operation, the programmer must create a *communicator* [20]. A communicator is a subset of the available GPUs that behaves as a fully connected clique, but in fact is backed by a network topology, such as a ring or tree, that is automatically configured by NCCL for efficient communication. GPUs within a clique are commonly referred to as *ranks*, and a GPU may appear as a rank within multiple communicators at once. In order to create a communicator, the programmer may generate a unique `id` for the communicator, then call `ncclCommInitRank` with `id` for each device to be included in the communicator. Each device must also be assigned a rank identifier local to that communicator, such that if there are n ranks in a communicator they are numbered from 0 to $n - 1$.

The library also facilitates point-to-point send and receive operations [21]. This communication is two-sided: for a data transfer to occur, the host must issue both a send command to the source rank, and a matching receive command to the target rank. Furthermore, multiple sends and receives may be combined together to allow the user to define their own collective operations beyond those provided. As with collective operations, all senders and receivers must be within the same communicator clique.

Groups and Command Matching. Beyond individual operations, NCCL provides a mechanism for batching several operations together, called *group calls*. The documentation states that groups are used for three distinct purposes [10] (labels added): “(1) managing multiple GPUs from one thread (to avoid deadlocks), (2) aggregating communication operations

to improve performance, or (3) merging multiple send/receive point-to-point operations.” Furthermore, it states that “All three usages can be combined together, with one exception: calls to `ncclCommInitRank()` cannot be merged with others.”

A related concept to groups is *command matching*. The library documentation describes several ways in which commands may be combined, including “aggregating” and “merging” (as seen above), as well as “fusion”. These terms are used without a clear prior definition, and it is at times unclear whether there is intended to be some overlap in their meanings, or if they are to be understood as referring to entirely separate concepts. For the avoidance of any confusion or disagreement with the documentation, we use a separate term not present in the documentation. We define *matching* as the process by which two or more communication commands, across one or more threads, are combined to form a single operation.

As a rule, communication commands must be matched in order to have well-defined semantics. One of the reasons why groups are so important is that commands issued from the same thread may only be matched if they are grouped together. Fig. 2 shows an illustration of the full command execution pipeline, including matching. In this example, each of three threads manages a single rank, and all three ranks are members of a single communicator clique. Each thread issues a `ncclBroadcast` to a stream on the corresponding rank. Once each copy of the command reaches the head of its stream, the commands are matched across all three ranks in the communicator, and the broadcast executes, transferring data between the three ranks accordingly.

Relation to MPI. The experienced reader may find a number of these concepts familiar from the Message Passing Interface (MPI) [22]. Ranks, communicators, collective communication, and two-sided point-to-point messages deliberately follow MPI in their design. Indeed NVIDIA makes clear that they intend NCCL as the GPU-oriented analogue of MPI [23].

Much work has already been dedicated towards the formalisation of MPI, including model-checking semantics for blocking and non-blocking operations by Siegel et al. [24], [25], a TLA+ specification of MPI 2.0 and the ISP dynamic verifier from Gopalakrishnan et al. [26], [27], [28], and protocol types for verifying collective matching due to López et al. [29].

A core element of work on formalising MPI is the non-determinism of receives using the wildcard `MPI_ANY_SOURCE`, which NCCL does not have. Rather, NCCL has its own unique challenges: characterising the interaction between host-side threads that issue commands, and device-side ranks that execute them; the relevance of CUDA Stream ordering semantics, as all commands must be first enqueued onto a stream before execution; and the introduction of command groups, which affects the matching and execution of commands. Therefore, whilst work on formalising MPI is relevant, none of it is directly analogous to the formalisation required for NCCL.

III. LITMUS TESTING FRAMEWORK

To rigorously investigate NCCL, and to ensure our claims are empirically backed, we have developed a litmus testing

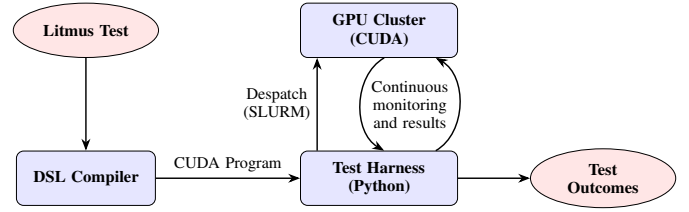


Fig. 3. Framework overview

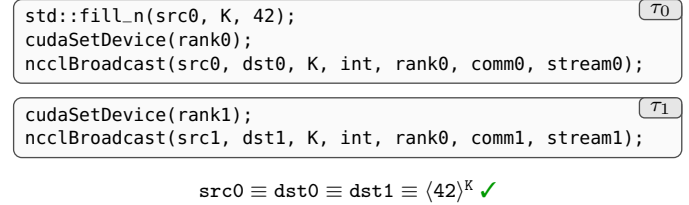


Fig. 4. Multi-threaded broadcast

framework with three goals: 1) express test cases concisely and clearly; 2) run tests repeatedly under consistent conditions to either observe unexpected behaviours or build confidence in their absence; 3) allow tests to capture a variety of configurations including various numbers of host threads, devices, and communicator configurations.

To achieve this, we created a domain-specific language (DSL) that captures the fragment of CUDA relevant to expressing programs using NCCL. Our DSL allows us to express test cases without the extensive boilerplate and environment setup required when writing them directly in CUDA. We also implemented a compiler that converts concise litmus tests expressed in our DSL to full CUDA test programs. Finally, we implemented a test harness in Python that launches tests, monitors their progress, captures results, and automatically checks whether their outcomes conform with the expected semantics. Fig. 3 shows how these components interact.

Our framework allows the expression of arbitrary test cases with respect to the number of threads and ranks, the number of streams and buffers allocated per rank, the size of allocated buffers, and the manner in which ranks are grouped into communicators. The tests in this paper were run using up to three NVIDIA A100 GPUs connected on a single node via PCIe, using CUDA version 12.9.0 and NCCL version 2.28.9. Further details, including how to replicate these experiments and a full specification of the DSL, are available in our artefact [11].

IV. LITMUS TESTS AND INSIGHTS

In these examples, we show relevant snippets from one or more program threads as distinct code blocks, labelled with τ_n to denote the n^{th} thread. We assume that variable names `srcN` and `dstN` are pointers to disjoint integer arrays of a fixed length K that are zero-initialised. Each example has exactly one communicator, with `commN` as the handle of rank N to that communicator, and these examples only use a single stream per rank, given by `streamN` for rank N . Where obvious, we elide these arguments from NCCL invocations. For litmus tests

```

std::fill_n(src0, K, 42);
ncclGroupStart();
  cudaSetDevice(rank0);
  ncclBroadcast(src0, dst0, ..., rank0, ...);
  cudaSetDevice(rank1);
  ncclBroadcast(src1, dst1, ..., rank0, ...);
ncclGroupEnd();

```

τ_0

$\text{src0} \equiv \text{dst0} \equiv \text{dst1} \equiv \langle 42 \rangle^K \checkmark$

Fig. 5. Broadcast using a group

```

cudaSetDevice(rank0);
ncclBroadcast(...);

```

```

cudaSetDevice(rank1);
ncclBroadcast(...);

```

τ_1

```

cudaSetDevice(rank0);
ncclAllReduce(...);

```

```

cudaSetDevice(rank1);
ncclAllReduce(...);

```

τ_3

Fig. 6. Potential command mismatch causes UB

with a well-defined outcome, we write the outcome below the example, using $\checkmark$ to indicate the outcome is valid, and listing statements that hold in that outcome separated by commas. In describing a program’s terminal state, we use $\equiv$ to denote element-by-element equality of arrays, and write $\langle v \rangle^N$ for the array of length N where each element holds value v .

We have validated the behaviour of these litmus tests in practice using our testing framework, running each test at least 100,000 times to attempt to catch any non-deterministic behaviour. Through conversation with the development team at NVIDIA, we have also confirmed that these results are intended to generalise to other hardware configurations.

Single- and Multi-Threaded Collectives. One of the simplest examples we could consider is a broadcast from one rank to a single peer. Fig. 4 shows how this could be done using one thread to manage each rank, while Fig. 5 shows how a group may be used to manage both ranks from a single thread.

In either case, the two broadcast commands are matched, by virtue of the fact that they are the same operation, operate on data of the same size and type, agree on the root rank, and target the same communicator. It should be noted that the use of a group is strictly required in the single-threaded example, as otherwise the call to rank 0 may be blocked waiting for the corresponding rank 1 call to match.

Command Matching. Importantly, note that collectives require *every* rank in the clique to participate; if we elided, in either example, the broadcast targeting rank 1, matching would fail, and the call to rank 0 would be blocked. In general, programs containing commands that might not match have undefined behaviour (UB), which is most commonly observed as a deadlocked state: this is because there may be a knock-on effect from the command remaining blocked which can cause commands issued towards the same communicator, stream or rank, or by the same thread, to fail to execute.

One execution of a program in which matching fails is sufficient for UB; in Fig. 6 the commands match so long as both τ_0 and τ_1 execute their commands before either of τ_2 or

τ_3 , or vice versa; however, in any other interleaving they will fail to match, so the program contains UB because in *at least one interleaving* matching fails.

Group Calls. We have reviewed the high-level characterisation of the functions of group calls, as identified in the documentation. Now we will clarify the low-level behaviour of groups. In short, groups disable blocking within a sequence of commands. This allows commands later in the group to become eligible for matching without depending on the completion of prior commands. Consider Fig. 7: data is passed from one rank to the next using point-to-point operations, with the final rank sending data to the first.

To achieve this pattern without using groups, it is necessary to strictly order all sends and receives to avoid a deadlock: for example, the order of the operations as written would lead to a deadlock in the absence of the use of groups, as each call to `ncclSend` would block, waiting for the corresponding `ncclRecv`; therefore no `ncclRecv` could ever be issued. The use of a group allows the commands to be freely rescheduled to prevent this deadlock from occurring.

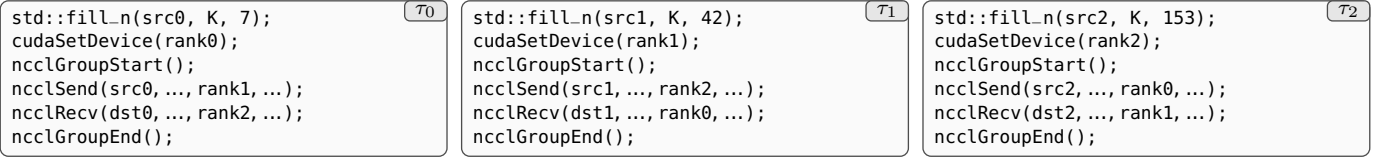
Group Ordering and Matching. Groups allow the *execution* of commands to be freely reordered, or executed concurrently, but do not affect matching, which must still occur in order. This distinction is not made in the documentation, but became apparent in our testing. Consider Fig. 8: if commands could be reordered for matching purposes, this would be a legitimate example, but in fact the documentation makes clear this program is not allowed, and in our discussions with NVIDIA they clarified that this example has undefined behaviour. The order of the commands in one of the threads must be reversed for the program to be well-defined.

It is, however, worth noting that this does not imply the stronger requirement that groups must contain an equivalent list of commands across threads. For example, Fig. 9 shows a broadcast called across three ranks in a communicator using two threads, where one issues a solitary broadcast command to one rank, and the other uses a group to issue the commands to the other two ranks. While it is not apparent from the documentation whether or not this is a legitimate way to invoke an operation, we found it to work in practice, and confirmed with NVIDIA that it is a supported pattern.

A consequence of the reordering permitted by groups is that if distinct commands access the same data, there will be a data race if at least one access is a write. It is not possible to enforce synchronisation within a group using `cudaStreamSynchronize` [10]. According to our discussions with NVIDIA, programs with data races have undefined behaviour: thus, due to a write-write race on `dst2`, the example in Fig. 10 has UB.

V. FUTURE WORK

This work lays a strong foundation for formalising NCCL. We have given intuitive descriptions of several core features of the library, which should be made rigorous and formal. Our testing framework can be scaled to a comprehensive test suite



$$dst0 \equiv src2 \equiv \langle 153 \rangle^K, dst1 \equiv src0 \equiv \langle 7 \rangle^K, dst2 \equiv src1 \equiv \langle 42 \rangle^K \checkmark$$

Fig. 7. Point-to-point data exchange

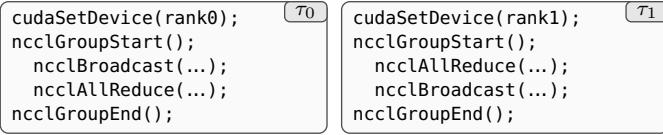
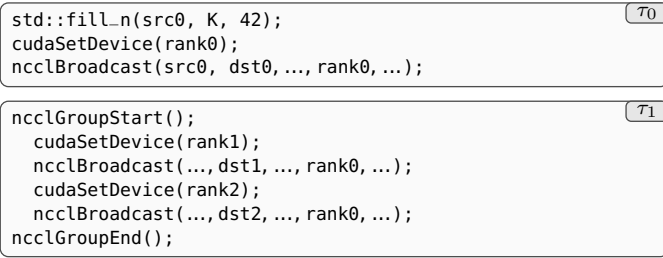


Fig. 8. Incorrect command order causes UB



$$src0 \equiv dst0 \equiv dst1 \equiv dst2 \equiv \langle 42 \rangle^K \checkmark$$

Fig. 9. Partial grouping of broadcast commands

that may identify whether there are any divergences between the behaviour intended by the developers of NCCL and the behaviour of its implementations in practice.

Our aim is that this work will support the development of a fully fledged operational semantics for programs written using NCCL and relevant parts of CUDA. A formal model would support further investigations into the reliability of NCCL. It would enable the construction of proofs of correctness for NCCL programs, and could be used as the basis for automated bug finding with strong guarantees of soundness or completeness. We also aim to expand the scope of our testing framework beyond multiple threads running on a single node, to arbitrary arrangements of processes running across nodes.

A semantics for NCCL could also support investigations into related libraries. AMD’s collective communication library

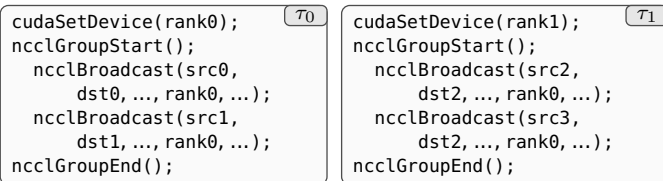


Fig. 10. Write-write data race on dst2

RCCL (ROCm Communication Collectives Library) is intended to be API-compatible with NCCL [30], so one may reasonably expect their semantics to be very similar. There are also a variety of other libraries that implement communication collectives, such as Microsoft’s MSCCL and MSCCL++ [31], [32], which may be characterised by a similar semantics.

REFERENCES

- [1] NVIDIA Corporation, “CUDA programming guide,” 2026. [Online]. Available: <https://docs.nvidia.com/cuda/cuda-programming-guide/index.html>
- [2] —, “NCCL (NVIDIA collective communications library) documentation,” 2025. [Online]. Available: <https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/>
- [3] PCI-SIG, “PCI Express Base specification revision 6.0 version 1.0,” PCI-SIG, Tech. Rep., 2022. [Online]. Available: <https://pcisig.com/pci-express-6.0-specification>
- [4] InfiniBand Trade Association, *InfiniBand Architecture Specification Volume 1, Release 1.2.1*, Beaverton, OR, 2007, available upon request.
- [5] B. Slechta, N. Comly, A. Eassa, J. DeLaere, and S. Raj, “NVIDIA NVLink and NVIDIA NVSwitch supercharge large language model inference,” NVIDIA Technical Blog, August 2024. [Online]. Available: <https://developer.nvidia.com/blog/nvidia-nvlink-and-nvidia-nvswitch-supercharge-large-language-model-inference/>
- [6] Axis Intelligence Research, “Nvidia statistics 2026: \$215.9B revenue & AI dominance data,” Axis Intelligence Research, 2026. [Online]. Available: <https://axis-intelligence.com/nvidia-statistics/>
- [7] PyTorch Contributors, “Distributed communication package – torch.distributed,” PyTorch documentation, 2026. [Online]. Available: <https://pytorch.org/docs/2.13/distributed.html>
- [8] TensorFlow Contributors, “tf.distribute.NcclAllReduce,” TensorFlow API documentation, 2026. [Online]. Available: https://www.tensorflow.org/api_i_docs/python/tf/distribute/NcclAllReduce
- [9] OpenXLA, “XLA: Accelerated linear algebra,” <https://openxla.org/xla>, 2024.
- [10] NVIDIA Corporation, “Group calls,” 2025. [Online]. Available: <https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/groups.html#group-calls>
- [11] M. Stuppel, A. F. Donaldson, and A. Raad, “NCCL Litmus Testing Framework,” Sep. 2026. [Online]. Available: <https://doi.org/10.5281/zenodo.22692769>
- [12] NVIDIA Corporation, “Thread block synchronization functions,” 2026. [Online]. Available: <https://docs.nvidia.com/cuda/cuda-programming-guide/05-appendices/cpp-language-extensions.html#thread-block-synchronization-functions>
- [13] —, “Warp synchronization function,” 2026. [Online]. Available: <https://docs.nvidia.com/cuda/cuda-programming-guide/05-appendices/cpp-language-extensions.html#warp-synchronization-function>
- [14] —, “Cooperative groups,” 2026. [Online]. Available: <https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/cooperative-groups.html#cooperative-groups>
- [15] —, “CUDA streams,” 2026. [Online]. Available: <https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/asynchronous-execution.html#cuda-streams>
- [16] —, “Stream synchronization,” 2026. [Online]. Available: <https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/asynchronous-execution.html#stream-synchronization>

- [17] —, “Explicit synchronization,” 2026. [Online]. Available: <https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/asynchronous-execution.html#explicit-synchronization>
- [18] Z. Hu, S. Shen, T. Bonato, S. Jauegy, C. Alexander, E. Spada, J. Dinan, J. R. Hammond, and T. Hoefer, “Demystifying NCCL: an in-depth analysis of GPU communication protocols and algorithms,” in *IEEE Symposium on High-Performance Interconnects, HOTI 2025, San Jose, CA, USA, August 20-22, 2025*. IEEE, 2025, pp. 48–59. [Online]. Available: <https://doi.org/10.1109/HOTI66940.2025.00024>
- [19] NVIDIA Corporation, “Collective operations,” 2025. [Online]. Available: <https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/collectives.html#collective-operations>
- [20] —, “Creating a communicator,” 2025. [Online]. Available: <https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/communicators.html>
- [21] —, “Point-to-point communication,” 2025. [Online]. Available: <https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/p2p.html#point-to-point-communication>
- [22] Message Passing Interface Forum, “MPI: A message-passing interface standard, version 4.1,” Nov. 2023. [Online]. Available: <https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf>
- [23] NVIDIA Corporation, “NCCL and MPI,” 2025. [Online]. Available: <https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/mpi.html#nccl-and-mpi>
- [24] S. F. Siegel and G. S. Avrunin, “Modeling wildcard-free MPI programs for verification,” in *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2005, pp. 95–106.
- [25] S. F. Siegel, “Model checking nonblocking MPI programs,” in *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, ser. LNCS, vol. 4349, 2007, pp. 44–58.
- [26] G. Li, R. Palmer, M. DeLisi, G. Gopalakrishnan, and R. M. Kirby, “Formal specification of MPI 2.0: Case study in specifying a practical concurrent programming API,” *Science of Computer Programming*, vol. 76, no. 2, pp. 65–81, 2011.
- [27] S. S. Vakkalanka, G. Gopalakrishnan, and R. M. Kirby, “Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings,” in *Proceedings of the 20th International Conference on Computer Aided Verification (CAV)*, 2008, pp. 66–79.
- [28] S. Vakkalanka, S. Sharma, G. Gopalakrishnan, and R. M. Kirby, “ISP: A tool for model checking MPI programs,” in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2008, pp. 285–286.
- [29] H. A. López, E. R. B. Marques, F. Martins, N. Ng, C. Santos, V. T. Vasconcelos, and N. Yoshida, “Protocol-based verification of message-passing parallel programs,” in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015, pp. 280–298.
- [30] Advanced Micro Devices Inc., “What is RCCL?” 2025. [Online]. Available: <https://rocm.docs.amd.com/projects/rccl/en/latest/what-is-rccl.html>
- [31] M. Cowan, S. Maleki, M. Musuvathi, O. Saarikivi, and Y. Xiong, “MSCCLang: Microsoft collective communication language,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS)*, 2023, pp. 502–514.
- [32] Microsoft Corporation, “MSCCL++,” 2025. [Online]. Available: <https://github.com/microsoft/mscclpp>