OpenCCL: A Proposed Open Standard and Reference Implementation Unifying *CCLs for Performance, Portability, and Productivity

Anthony Skjellum
ASCEND and Dept. of Computer Science
Tennessee Technological University
Cookeville, TN, USA
askjellum@tntech.edu

Purushotham V. Bangalore
Dept. of Computer Science
University of Alabama
Tuscaloosa, AL, USA
pvbangalore@ua.edu

Thomas Hines, Kayla Humphrey,
Javontae Martin, Amr Akmal Abouelmagd,
Riley Shipley, and Rajat Bisht
ASCEND and Dept. of Computer Science
Tennessee Technological University
Cookeville, TN, USA

Zulkar Nine
Arkansas Tech University
Russellville, AR, USA
mnine@atu.edu

I. The Problem

The `*CCL` is the communication layer under a distributed AI framework: point-to-point, collective, and accelerator-optimized. NCCL [1], the de-facto CCL, is modeled on MPI: NVIDIA's GPU APIs, ported as AMD's RCCL [2] and forked by vendors (Meta's NCCLx/RCCLx [3]). NCCL is a toolkit, not a specification: thin semantics, much unsaid, and not vendor-neutral. The API advances when NVIDIA chooses (including changes of behavior and deprecation), and others reimplement, update, and/or fork. New fabrics, accelerators, and workloads currently focus on baseline-NCCL portability; extension libraries have good abstractions but no broad platform support; and there is no community path from best practice to supported feature. The result is vendor lock-in and limited creativity for applications and the community. Further, specialized features that differentiate alternative hardware platforms may not be made available to applications.

II. OpenCCL

OpenCCL™ is a NCCL-API-upward-compatible re-expression of NCCL and its variants, and a proposed vendor-neutral standard for performance, portability, and productivity. OpenCCL seeks to follow the successful model of the MPI Forum and the MPICH reference implementation [4], [5], under which applications became portable and software investments survived generations of machines while vendors specialized behind a stable interface. OpenCCL is foremost an **open standard**—a normative header, `openccl.h`, under an implementation-independent `occl*` namespace, with a complete interposition API/ABI for tools such as Caliper and TAU—and second a performant **reference implementation** that vendors and the community may specialize. Programs built against `nccl.h` recompile unchanged. At present, OpenCCL addresses all 33 entry points of NCCL 2.23.4, its first target, and 53 of NCCL 2.30.7's 83. A build may also masquerade as an NCCL or RCCL soname for binary drop-in.

III. Why now

A vendor can change or break its interface at will. For fast-moving AI work that can be acceptable; for long-lived and costly software it is a risk. OpenCCL offers a choice: the most-stable or the newest level, an experimental tier for pre-standard features, and application-specific profiles for greater, focused optimization. Agile in concept, unlike the MPI Forum, OpenCCL as a standard will have an on-ramp and an off-ramp that MPI lacks.

IV. Architecture

OpenCCL's collectives are enabled by *RAPIDS* [6], in particular its kernel-initiated **Cascade** APIs on Slingshot-11. RAPIDS supplies bring-up, persistent flow-controlled channels, and operations; OpenCCL supplies the NCCL API (and more), the collective algorithms, and their kernels. OpenCCL offers fast transfers, with the middle level code removed; it leverages RAPIDS, which lightly abstracts the fabric's own API—`libcxi` on Slingshot-11—rather than a portability layer such as libfabric, which hides network-specific features. Cascade is working and kernel-initiated on Slingshot-11, copies GPU to GPU on one node, and utilizes a stream-triggering interface that keeps NCCL semantics. Codesign is bilateral: OpenCCL is presently Cascade's key consumer, and each codebase has motivated features and functions in the other. OpenCCL will utilize different RAPIDS mechanisms on alternative networks, potentially, and builds on top of CPU and MPI drivers for portability.

V. Implementation

OpenCCL uses C++20, requiring only CMake and pthreads; accelerator runtimes, bootstrap channels, and vendor libraries resolve by `dlopen` at run time. State-of-the-art collective algorithms are in progress, importing the best ideas from NCCL, NCCLx, MSCCL, and MPI Advance. MSCCLang's AllToNext [7], with its optimized on-node/off-node algorithm, is implemented as an `occl` feature and runs on Cascade as a device kernel.

VI. Correctness and results

Current test platforms include LLNL Tuolumne and Matrix, Purdue Anvil, UIUC Delta, and a two-node DGX Spark cluster. On Tuolumne (MI300A, Slingshot-11), Cascade's collectives are validated from 8 B to 256 MiB at up to 4 nodes $\times$ 4 GPUs. OpenCCL runs real AI software unmodified. For instance, PyTorch [8] trains end-to-end on three parallelism strategies—data-parallel with ranks bit-identical, sharded (FSDP), and tensor-parallel via Megatron-LM. With our Rally benchmark (a GPU-to-GPU ping-pong with a trivial kernel between ping and pong), Cascade demonstrates superior performance compared to Cray MPICH, which is poorly tuned on-node above 1 KB. For Allreduce at 128 ranks (32 nodes $\times$ 4 GPUs, a trivial kernel between each Allreduce), OpenCCL already outperforms RCCL and Cray MPICH on Slingshot-11 except below 256 bytes, with the best configurations used for RCCL and MPICH. These are just preliminary results for OpenCCL.

VII. Principled use of AI

OpenCCL is built by plan-heavy, prompt-driven engineering followed by programmer optimization and refactoring. The authors supply design, functional, and non-functional requirements—a "mixture of human middleware designers"—and drive iteration. Design documents are written and updated as we build, so each increment refines a recorded design driven by performance and functionality goals. Experience with actual low-level behavior is captured as tests run on diverse HPC systems, and codesign spans two codebases, OpenCCL and RAPIDS. A taxonomical study of 24+ `*CCL` variants worldwide identified unique features and commonality as input toward a standards-driven, performance-portable programming model.

References

[1] NVIDIA, "NCCL: NVIDIA collective communications library," https://github.com/NVIDIA/nccl, 2024, accessed 2026.

[2] AMD, "RCCL: ROCm communication collectives library," https://github.com/ROCm/rccl, 2024, accessed 2026.

[3] Meta, "torchcomms: NCCLX, RCCLX, and CTran," https://github.com/meta-pytorch/torchcomms, 2026, open-sourced NCCLX (NVIDIA) and RCCLX (AMD) extensions of NCCL/RCCL; see also *Collective Communication for 100k+ GPUs*, arXiv:2510.20171.

[4] Message Passing Interface Forum, "MPI: A message-passing interface standard, version 4.0," University of Tennessee, Knoxville, Tech. Rep., 2021, https://www.mpi-forum.org/docs/.

[5] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, 1996.

[6] R. Shipley, A. Skjellum, P. V. Bangalore, and P. G. Bridges, "RAPIDS: Reduced API data-transfer specifications," in *Research Poster, International Conference for High Performance Computing, Networking, Storage and Analysis (SC24)*, 2024, Tennessee Tech / Alabama / UNM. A divide-and-conquer decomposition of all-purpose communication libraries into small, focused APIs; the Channel API removes matching from the persistent path.

[7] 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)*. ACM, 2023, pp. 502–514.

[8] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, and S. Chintala, "PyTorch distributed: Experiences on accelerating data parallel training," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3005–3018, 2020.