HOSTCCL: DECOUPLING COLLECTIVE PROGRESS FROM GPU EXECUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

GPU-driven collective communication uses streaming multiprocessors (SMs) for protocol progress as well as tensor processing. Peer-readiness waits can retain GPU resources while data processing is stalled. We present HostCCL, a collective runtime that moves dependency progression to the CPU and uses copy engines for data movement. A CPU executor tracks tile dependencies and submits only ready reductions to a GPU worker, allowing transfer concurrency and reduction parallelism to be configured independently. Batched submission and ordered completion feedback let completed tiles begin forwarding while later reductions execute. On four and eight PCIe-connected L40S GPUs, HostCCL runs AllGather and AllToAll without steady-state GPU kernels, and ReduceScatter and AllReduce with a one-SM worker budget per GPU. For the tested large FP32 reductions, HostCCL uses a 50–75% smaller SM budget than default NCCL and improves throughput by up to 23.3%. In a controlled four-GPU experiment pairing 256 MiB AllGather with fixed GEMM work, HostCCL completes the joint work in 8.7 ms, compared with 14.9–15.2 ms for default NCCL.

1 INTRODUCTION

Distributed machine-learning workloads use collectives to exchange activations, combine gradients, and redistribute model state (Shoeybi et al., 2019; Li et al., 2020b; Rajbhandari et al., 2020). Collective kernels share streaming multiprocessor (SM) resources with model computation. When communication and computation overlap, the resources assigned to collectives can limit the capacity available to model kernels. We study how to sustain collective throughput with a small SM budget, leaving more execution capacity available for computation.

GPU-driven implementations use communication kernels to process tensors and advance communication protocols. NCCL's conventional paths organize work into channels serviced by GPU thread blocks (NVIDIA, 2026e;h). MSCCLang compiles collective schedules for a GPU runtime whose blocks execute primitives and synchronize through device-memory semaphores (Cowan et al., 2023). ResCCL improves primitive scheduling and reuses blocks across communication operations within GPU kernels (Liu et al., 2025). In these execution paths, GPU resources support dependency checks as well as data processing; waiting for a peer can retain those resources even when tensor work cannot proceed. Resource tuning adjusts the GPU allocation, and copy engines (CEs) can offload payload movement (Liu et al., 2025; Jeaugey et al., 2025). A CPU executor could track unresolved dependencies and release ready operations without assigning GPU blocks to peer-readiness waits.

We present *HostCCL*, a collective runtime that advances dependencies on the CPU, moves data through CEs, and submits only ready reductions to a configurable GPU worker. A CPU executor tracks communication and reduction dependencies over tensor tiles. The CPU handles control metadata; tensor payloads move between GPU buffers. This separates transfer concurrency from reduction parallelism: the number of in-flight transfers can grow without increasing the worker's SM budget. Movement-only collectives need no worker. Reducing collectives use a small worker grid sized for the ready arithmetic supplied by the transport. The design targets collective execution with zero or a few SMs, with the reduction budget chosen for the platform and workload.

Keeping a small worker productive requires prompt CPU–GPU coordination. The executor submits reductions as their inputs arrive and releases downstream transfers as reductions complete. HostCCL reuses compiled tile dependencies and batches available tasks without waiting for a batch to fill.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Ordered completion feedback lets each completed tile release its successors while later reductions execute, so forwarding does not wait for the worker kernel to exit. Explicit retirement drains accepted work before worker exit, while visibility and stream-ordering rules preserve safe buffer access and the caller's execution order.

We evaluate HostCCL on four and eight PCIe-connected L40S GPUs against default NCCL and NCCL under matched SM budgets. HostCCL executes AllGather and AllToAll without GPU workers and ReduceScatter and AllReduce with a one-SM budget per GPU. For the tested large FP32 reductions, throughput exceeds default NCCL by up to 23.3%. A controlled four-GPU experiment pairing AllGather with fixed GEMM work also shows shorter joint completion time than default NCCL.

Our contributions are:

- A collective execution model that keeps remote-readiness dependencies on the CPU and configures transfer concurrency independently of GPU reduction parallelism.
- A CPU–GPU progress interface that batches ready work and uses ordered task completions to release downstream transfers, with explicit worker retirement and stream-ordering rules.
- An evaluation relating complete-collective throughput to GPU SM budgets on PCIe L40S GPUs, with default and fixed-budget NCCL comparisons and a controlled compute–communication overlap experiment.

2 Background and Motivation

Collective operations and dependencies. AllGather and AllToAll move data among participating ranks; ReduceScatter and AllReduce also combine inputs through a reduction operator. Consider one data segment at an intermediate rank of ring ReduceScatter (Patarasuk & Yuan, 2009). The preceding rank supplies a partial result P, this rank contributes X, and the next rank receives $P + X$. Reduction requires visible inputs; forwarding requires the completed result and a writable destination. Different segments can occupy different stages of this path, allowing transfers and local arithmetic to overlap.

GPU resources for protocol progress. In GPU-driven paths, communication kernels enforce these dependencies as well as process data. NCCL's Ring/Simple path partitions channel data into chunks and processes slices within them. Its `recvReduceSend` primitive combines receive, reduction, and forwarding, with GPU threads checking peer progress and buffer availability alongside data processing (NVIDIA, 2026g;h). While a peer dependency remains unresolved, the responsible thread block can retain execution resources even though its data operation cannot proceed. This allocation supports protocol waiting as well as useful tensor work, and can reduce the capacity available to concurrent model kernels.

Moving dependency progression to the CPU. CEs can transfer payloads between GPU buffers without SM copy kernels (Jeaugey et al., 2025). The CPU can track unresolved communication dependencies, observe readiness and completion, and release operations when their prerequisites are satisfied. Moving these readiness checks and dependency updates to the CPU would leave CEs to move data and a small GPU worker to execute ready arithmetic. Movement-only collectives would need no worker; reducing collectives would use a grid sized for their arithmetic. GPU-local coordination and memory ordering remain necessary for task execution and result publication. The runtime must preserve these dependencies while advancing transfers and reductions promptly.

3 Design

HostCCL executes a collective as a dependency graph of tile operations. The CPU advances this graph, the transport moves tensor data, and a small GPU worker executes ready reductions. Figure 1 shows how these components process a ReduceScatter tile from arrival through reduction and forwarding.

3.1 Execution model and tiled plans

Each rank has a CPU executor and three backend interfaces: transport for movement, signal for peer notifications and waits, and device for local data operations. A collective submission creates a *run* of a reusable *plan*. The plan is an immutable dependency graph with command templates; the run

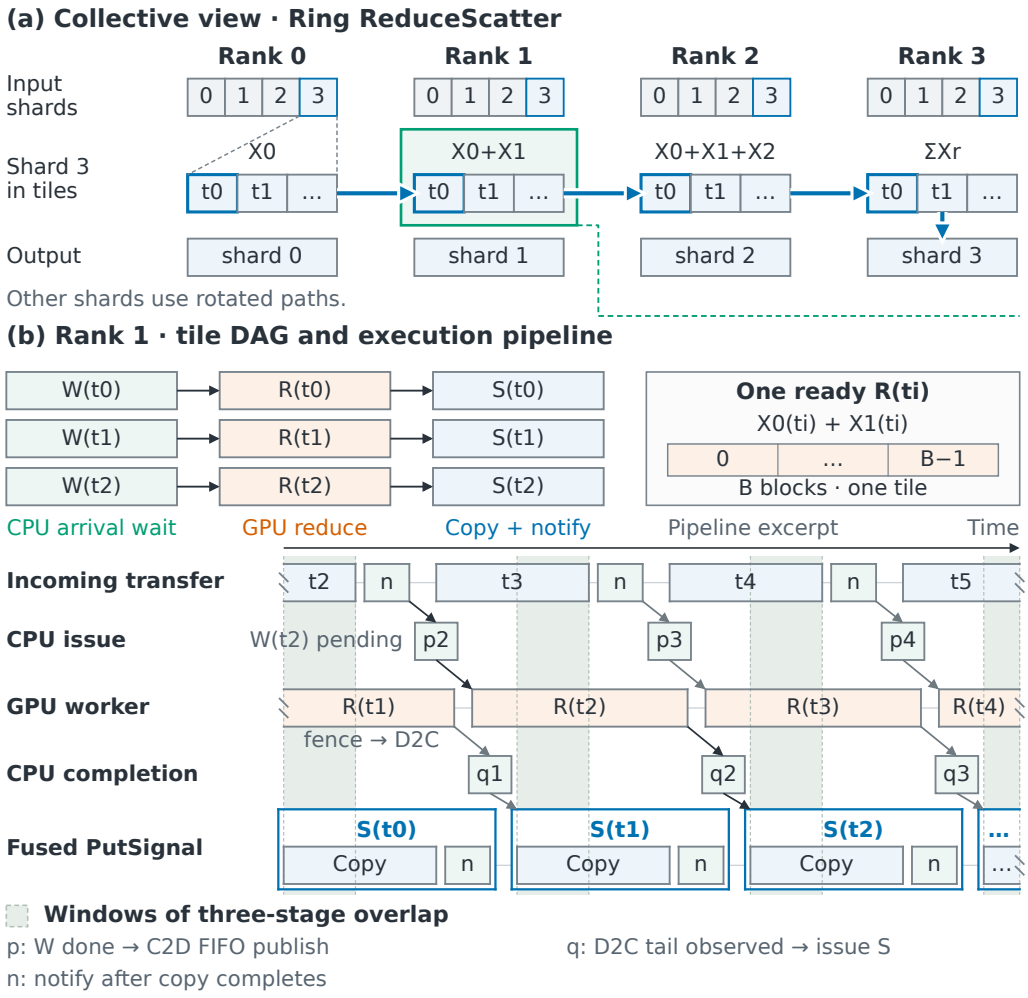

Figure 1: HostCCL's tile pipeline for ring ReduceScatter. (a) Shard 3 accumulates along ranks $0 \to 1 \to 2 \to 3$. (b) At rank 1, CPU arrival waits release reductions, whose completion releases forwarding. Different tiles overlap receipt, reduction, and forwarding; shading marks three-stage overlap. Each S_i integrates copy and notification. All B blocks cooperate on one FIFO task at a time ($B = 1$ in the main L40S configuration). The timeline is schematic; worker gaps do not imply exit.

supplies the invocation's buffer bindings, protocol epoch, and mutable progress state. The executor advances a run once its stream inputs and temporary GPU storage are available. Tensor payloads stay in GPU memory and on the transport path.

Construct the tile DAG. An algorithm builder emits rank-local transfers, receives, and receive–reduce operations, each with a peer, byte range, and predecessors. Lowering partitions these ranges into contiguous tiles of size τ, with a shorter final tile. Matched sends and receives use the same partition, and corresponding producer/consumer tiles retain their dependency edges. A tile is a data range, not a device task: several operations can act on that range as it moves and accumulates.

A receive lowers to a CPU arrival wait; receive–reduce adds a device reduction after that wait. A transfer becomes `PutSignal`, which moves its tile and notifies the receiver. An intermediate ReduceScatter hop thus has the chain $W_i \to R_i \to S_i$: arrival wait, reduction, and forwarding of tile i. The plan precomputes successor lists, dependency counts, and initially ready nodes. Each run has its own counters and ready queues, so repeated calls share the graph without sharing progress state.

Figure 1 follows one shard across ranks and expands an intermediate hop. When tile i arrives, the CPU submits R_i to the worker. Its task completion releases S_i without waiting for the worker kernel

to exit. Meanwhile, other tiles can arrive or be forwarded. Reductions within one FIFO execute in order, as do copies on a peer stream; different stages process different tiles concurrently. AllGather and AllToAll use movement and notification operations without reduction tasks and therefore need no worker during steady-state calls.

Compose collective phases. The ring AllReduce builder connects each shard's ReduceScatter producer to its AllGather consumers. Lowering preserves these tile edges, allowing dissemination of a completed range without a whole-phase barrier. This composition exposes overlap within one collective invocation; it does not merge independent calls. Appendix F.1 gives the lowering and tiling rules.

3.2 Host progress and transport

Release ready successors. Producer completion releases its successors; a node with several predecessors becomes ready when its remaining count reaches zero. For successors with a single predecessor, the plan compiles the fan-out into a *ready segment*: completion publishes one record, which dispatch expands into commands. The executor records the shared readiness once and dispatches the successors as separate operations.

The executor batches currently ready commands by backend, without waiting for more tiles to arrive. Before making a command visible, it records the run and operation that own its completion. Backends accept a prefix; unaccepted commands remain pending for retry. Backpressure from one peer does not prevent attempts to dispatch to other peers. Unresolved arrival waits stay in CPU state: the device FIFO contains ready arithmetic, with no task waiting for a remote transfer. Appendix F.2 details completion ownership and retries.

Fuse transfer and notification. A `PutSignal` request combines a data transfer and its receiver notification. The transport adapter observes copy completion before publishing the readiness tag, avoiding a separate notification command dispatched by the executor. The receiver's CPU wait completes when the matching tag arrives, releasing the tile's next operation. Per-peer copy windows keep multiple transfers in flight; control queues advance notifications independently of unrelated copies.

Dependencies connect operations across the three backend interfaces. Standalone notifications and arrival waits use the signal interface; fused notifications execute inside the transport adapter. A device completion releases a local result's transfer, whose later notification establishes readiness at the destination. Local task completion and peer arrival are distinct graph dependencies (Appendix F.3).

3.3 CPU–GPU interface and reduction worker

A worker *generation* is one kernel launch with a fixed grid of B blocks, which sets its SM budget. A CPU-to-device (C2D) FIFO carries ready-task descriptors; a device-to-CPU (D2C) completion tail reports finished tasks. This asymmetric interface carries control information, not tensor payloads.

Publish ready tasks. Descriptors and separate argument slots reside in GPU memory mapped for CPU writes; the CPU-owned head is in pinned host memory. For R_i, the arguments identify the received partial and matching local input to reduce, the tile length, and the result destination. The CPU makes the descriptors and arguments visible before publishing the head, so the worker sees a complete task prefix. The first task is published before launch. While a generation is open, a bounded batch of available tasks can share one head publication without waiting for additional arrivals.

Cooperate within a task. All B blocks consume the same task and reduce disjoint portions of its data. They finish that task before advancing to the next; blocks do not dequeue unrelated tasks. Block 0 reads the published head and shares a device-resident hint, allowing the grid to drain the observed prefix before polling again. Vectorized operations and independent loads increase useful work per block. At $B = 1$, inter-block coordination disappears; local barriers and output ordering remain. The evaluated path uses one FIFO; additional FIFOs would have separate grids that also count toward the SM budget.

Eligible final reductions also store their result directly in application output, avoiding a separate scratch-to-output copy. Successors reference the actual result location. This combines reduction and final placement within a task, subject to the layout conditions in Appendix F.1.

Release successors at task completion. All blocks finish their result writes before the worker publishes its GPU-resident completion tail. Because tasks finish in FIFO order, this monotonic tail replaces a queue of per-task response messages. The GPU publishes progress after every task. The CPU reads the tail through its mapping, associates the completed sequences with their graph operations, and retires that prefix. One observation can retire several accumulated completions. Completion processing does not wait for a full batch, so R_i can release S_i while later reductions execute. Queue ordering and slot reuse are specified in Appendix F.4.

Set the SM budget independently of transport concurrency. Tile size τ, the per-peer copy window, and B control different resources. Smaller tiles release downstream work earlier but require more commands and notifications. A larger copy window exposes more transfers without enlarging the worker; increasing B adds arithmetic parallelism without changing the transfer-dependency rules.

The target supply of reduction work depends on interconnect topology and GPU placement (Li et al., 2020a). For a fixed datatype and fan-in, let R_{target} denote this target, chosen independently of throughput observed at the current worker budget, and let $R_{\text{worker}}(B)$ denote capacity with ready inputs. Both rates count the same reduction work per second. The configuration criterion is a small SM budget B satisfying $R_{\text{worker}}(B) \geq R_{\text{target}}$. Sustaining collective throughput also requires timely CPU dispatch and transport progress.

3.4 Worker lifetime and correctness

HostCCL distinguishes three completion points. A task's completion releases graph successors; kernel exit permits reuse of generation state; completion of all run operations releases the caller's stream. The CPU detects task and run completion from operation feedback, independently of kernel-exit events.

CPU-controlled retirement. A dispatch pass leaves a generation open if it submits device work or has ready device work deferred by backpressure. Otherwise, it appends `EndBatch` to seal the generation. One FIFO slot is reserved for this marker. All accepted tasks precede it, so they drain before every block returns. Sealing is irreversible: later work opens a successor generation on the same worker stream, ordered after its predecessor. The kernel uses no idle timeout. Outstanding transfers can continue after worker exit and release reductions for a later generation.

Preserve stream order. Submission records input readiness and installs a completion gate on the caller's stream. The executor admits the run when both its inputs and scratch resources are ready, and opens the gate when the run finishes. Calls on the same stream retain submission order; independently ready streams progress on a best-effort basis.

Authorize writes before announcing arrival. If ranks reach successive calls at different times, a sender could overwrite a scratch lane still owned by an earlier run at the receiver. A notification after the copy cannot prevent that overwrite. HostCCL therefore requires receiver admission before the sender moves data: the receiver must own the scratch storage and have ready stream inputs before publishing admission. After the permitted copy completes, `PutSignal` announces the tile's arrival. Admission protects destinations from premature reuse; arrival releases tile consumers. Both waits remain on the CPU, and independently admitted work can progress without an all-rank barrier.

Order visibility and reuse. Transfer visibility precedes arrival, arrival precedes reduction, and result visibility precedes completion feedback. Per-run epochs distinguish repeated calls; buffers remain live while consumers can access them. The GPU worker still polls its task queue and uses local barriers and memory ordering during execution. Appendices F.5 and F.6 specify generation and buffer lifetimes and publication contracts.

4 Implementation

HostCCL is implemented in approximately 24,000 lines of C++ and CUDA. The code comprises a collective layer (APIs, plans, and executor), transport backends, and a GPU worker. NCCL-compatible entry points support AllGather, AllToAll, ReduceScatter, and AllReduce; the evaluated reducing collectives use ring schedules.

In the evaluated CUDA path, CEs move payloads through IPC mappings (NVIDIA, 2026a). GDRCopy maps GPU-resident task descriptors, arguments, and completion tails for CPU access (NVIDIA, 2026c); FIFO heads use pinned host memory. Registrations and IPC mappings are cached and prepared before application-stream completion gates are installed. This ordering prevents mapping setup from waiting behind a gate whose transfer requires that mapping. Initialization returns after an empty warmup worker drains. Steady-state AllGather and AllToAll use CE movement and host progress without launching a worker.

The executor uses one dispatch thread and separate device, transport, and signal completion threads; IPC adapters also run progress threads. Their scheduling determines when arrivals and completions release subsequent operations. The launcher sets each rank's GPU-local CPU-affinity mask before initialization, so runtime threads inherit the allowed cores. Threads share these cores under OS scheduling. Appendix F.7 details thread and memory placement; Section 5.1 specifies the CPU allowances and SM budgets used in evaluation.

5 EVALUATION

We evaluate complete-collective throughput, performance under matched SM budgets, and interference with concurrent computation. Experiments cover four collectives on four and eight PCIe-connected L40S GPUs, with HostCCL using a zero- or one-SM budget in the main comparison.

5.1 METHODOLOGY

Platform and configurations. Our server has eight L40S GPUs and 64 physical CPU cores. Four GPUs attach to one NUMA domain; eight span both. Before initialization, each rank receives a CPU-affinity mask covering eight GPU-local physical cores. Masks are disjoint across ranks and identical for both libraries; threads share their allowed cores under OS scheduling. We use CUDA 13.3 and NCCL 2.31.2. NCCL default leaves CTA, algorithm, protocol, and efficiency-policy settings unmodified. HostCCL uses one FIFO, one IPC stream per peer, ring reducing schedules, and a one-SM worker budget; movement collectives bypass the worker. The size sweep uses out-of-place, ordinary CUDA allocations without explicit native buffer/window registration. NCCL's documented intra-domain zero-CTA CE path targets NVLink, unlike this PCIe platform (NVIDIA, 2026j).

Measurements. Reductions use FP32 Sum. Size S denotes full per-rank input for ReduceScatter/AllReduce, output for AllGather, and send volume for AllToAll. We time complete calls, including submission, host progress, data movement, reduction where needed, and completion. The size sweep synchronizes every call. We report slowest-rank wall time and nccl-tests bus bandwidth (NVIDIA, 2026i), using medians and min–max ranges over three independent process launches. Changing-input stream-consumer checks and warmups precede timing, and library order reverses in the second repetition.

SM budgets. We set the communication SM budget through the number of communication blocks: B blocks use at most B SMs concurrently. This budget applies to communication kernels; application kernel grids are not interpreted as SM counts. Separate kernel traces verify communication-block counts, while performance timings come from unprofiled runs. Appendix A provides the full protocol, trace procedure, and native registration controls.

5.2 COMPLETE COLLECTIVES WITH ZERO OR ONE SM

Figure 2 pairs throughput with the communication SM budget across eight sizes from 4 KiB to 256 MiB. HostCCL launches no steady-state GPU kernels for AllGather or AllToAll and uses a one-SM budget for ReduceScatter and AllReduce throughout the sweep. With this budget, AllGather and both reductions exceed default NCCL throughput at every tested size from 64 to 256 MiB on both four and eight GPUs, with gains of 13.6–23.3%.

Higher reduction throughput with a smaller SM budget. Default NCCL has an SM budget of four on four GPUs and two on eight at the large-message points. HostCCL improves reduction throughput with a 75%/50% smaller SM budget, respectively. At 256 MiB, its four-GPU ReduceScatter and AllReduce gains are 19.2% and 18.6%; the eight-GPU gains are 23.3% and 18.7%. For example,

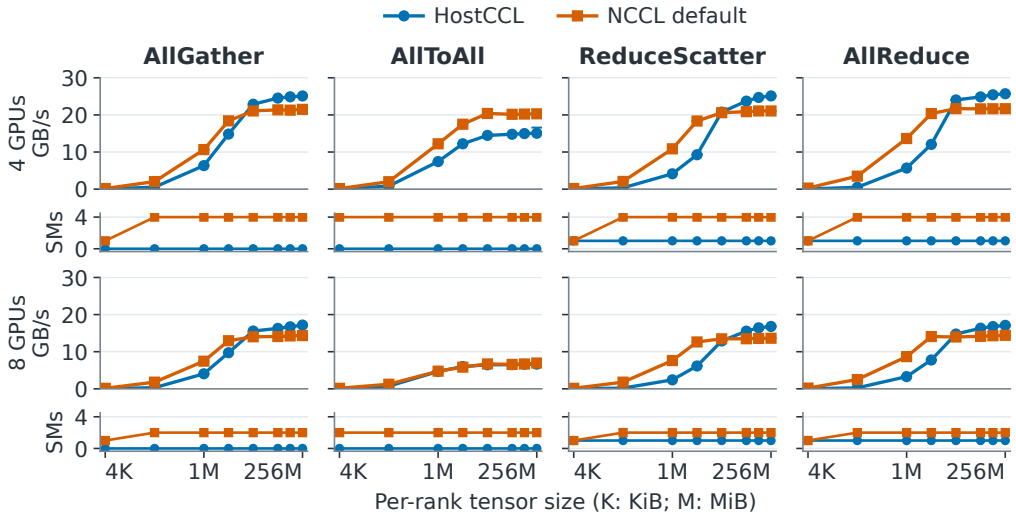

Figure 2: End-to-end collective throughput and SM budget on four/eight L40S GPUs. HostCCL uses a zero-SM budget for AG/A2A and one SM for FP32 RS/AR; NCCL uses default tuning. Resource strips report SM budgets from communication-block counts, independently traced on every rank and size. Points are medians and min–max of three launches with per-call synchronization and identical CPU allowances.

eight-GPU ReduceScatter reaches 16.80 GB/s with a one-SM budget, versus 13.63 GB/s with NCCL's two-SM budget.

Movement without SM execution. AllGather combines a zero-SM budget with 16.8%/19.7% higher throughput at 256 MiB on four/eight GPUs, reaching 25.10/17.17 GB/s. AllToAll also eliminates worker kernels at every tested size. At 256 MiB, it retains 74.2%/95.7% of default NCCL's throughput on four/eight GPUs, while NCCL has four-/two-SM budgets.

AllGather and AllReduce exceed NCCL from the tested 16 MiB point on both GPU counts, and both reductions do so consistently from 64 MiB. NCCL remains faster for AllGather and the reductions at 4 KiB–4 MiB.

5.3 Throughput under a fixed SM budget

We test whether a smaller NCCL SM budget can match HostCCL's one-SM throughput. This separate eight-GPU sweep synchronizes every four calls. Native runs set `NCCL_MIN_CTAS=NCCL_MAX_CTAS=B`, retaining automatic algorithm/protocol selection; default and efficiency policies provide additional references. HostCCL retains its ring schedule.

Matched-budget throughput. With a one-SM budget, HostCCL delivers 16.38 GB/s for 128 MiB ReduceScatter and 17.11 GB/s for 256 MiB AllReduce: 40.0% and 44.9% above NCCL at the same budget. Its one-SM throughput also exceeds every tested native budget up to 64 SMs. Increasing HostCCL's SM budget produces little additional collective throughput.

Reduction capacity. The right panel isolates worker service with ready inputs. Each GPU receives 112 MiB of reduction payload in 2 MiB tasks; the two input arrays total 224 MiB, exceeding its 96 MiB L2. Tasks are supplied before device timing. Increasing the SM budget from one to sixteen raises worker payload throughput from 23.46 to 147.23 GB/s, while the complete collectives above gain little. These service rates measure payload per device time, distinct from collective bus bandwidth. A small SM budget is sufficient for these collective paths. Tile-level traces separate completion-detection delays from forwarding queues (Appendix C).

CPU provisioning. CPU threads keep transport and the worker supplied with ready operations. For eight-GPU 128 MiB ReduceScatter, four local physical cores per rank sustain the throughput seen with eight; the main experiments retain an eight-core allowance across sizes. At this allowance and a

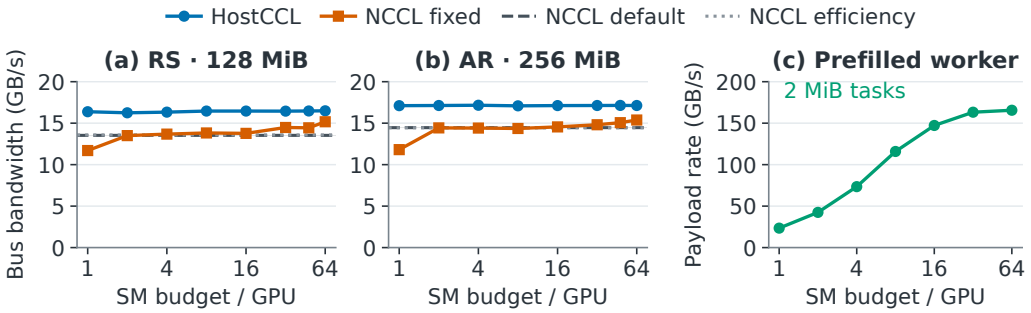

Figure 3: Eight-L40S SM budget and reduction capacity. Left/middle: FP32 collective bus bandwidth, synchronizing every four calls; native policy references are distinct from fixed-budget points. Right: prefilled worker payload rate with 2 MiB tasks, excluding collective transport. All panels show three-launch medians and min–max. Collective bus bandwidth and worker payload rate use different work and timing definitions.

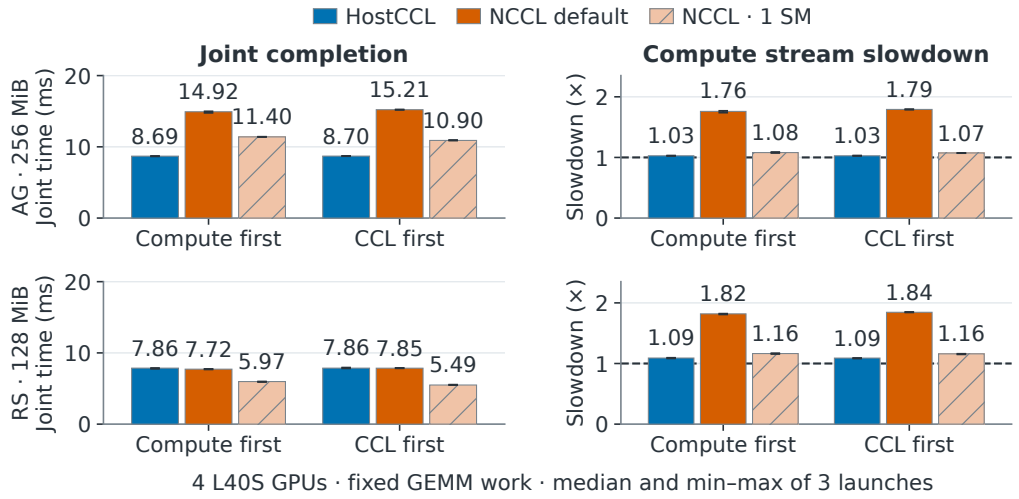

Figure 4: Controlled compute–communication overlap on four L40S GPUs. Left: slowest-rank joint wall time. Right: compute-stream duration relative to the matching compute-only run. Each iteration completes the same GEMM work and one collective. Both submission orders are shown; medians and min–max over three launches. HostCCL uses zero-/one-SM budgets for AG/RS.

one-SM HostCCL budget, the 128 MiB ReduceScatter and 256 MiB AllReduce points consume 5.6–5.8 times default NCCL's summed CPU-ms/call. Appendices D–E report the placement, allowance, and CPU-time measurements.

5.4 Compute–communication overlap

We measure compute-stream slowdown and joint completion during concurrent communication and computation. Each of four ranks runs one collective and fixed GEMM work on two nonblocking, equal-priority streams with disjoint buffers. Standalone GEMM timing selects sixteen/eight 4096^3 GEMMs for 256 MiB AllGather/128 MiB ReduceScatter, with FP16 inputs and FP32 accumulation/output. Compute-only, CCL-only, sequential, and concurrent modes use identical work across libraries. Both concurrent submission orders use the CPU allowances in Section 5.1.

Worker-free AllGather benefits joint execution. Figure 4 shows joint completion in 8.69–8.70 ms with HostCCL, versus 14.92–15.21 ms with default NCCL and 10.90–11.40 ms with one-SM NCCL.

The improvement holds for both submission orders. HostCCL's compute stream remains within 3% of its standalone duration, compared with a 76–79% slowdown under default NCCL.

One-SM reduction limits compute interference. With ReduceScatter, HostCCL limits compute slowdown to 9%, versus 82–84% under default NCCL. Its communication duration grows under concurrency, leaving joint time near default NCCL and above one-SM NCCL. Both collectives reduce interference with GEMM; AllGather also shortens joint execution. The measured timings reflect GPU execution, memory contention, and submission timing together. Appendix B gives the full protocol and per-stream results.

6 Related Work

Resource allocation. NCCL offers CTA budgets and conditional zero-CTA paths for reduction-free collectives (NVIDIA, 2026d;e;j; Jeaugey et al., 2025). AutoCCL tunes NCCL's algorithms, protocols, and resources (Xu et al., 2025); ResCCL improves primitive schedules and block assignments within communication kernels (Liu et al., 2025). HostCCL keeps remote-readiness waits on the CPU and sizes the GPU worker for arithmetic, configuring its SM budget separately from transfer concurrency.

Host progress. VCCL combines CPU progress, CE transfers, and stream dependency management for reduction-free P2P communication (Zhang et al., 2025). UCCL-Tran implements host transport control beneath collective libraries (Zhou et al., 2026). BytePS places gradient aggregation on CPUs (Jiang et al., 2020). HostCCL's CPU executor advances the collective's tile DAG, including receive–reduce–forward dependencies, and submits only ready reductions to a small GPU worker.

Algorithms and programming. SCCL and TACCL synthesize collective schedules (Cai et al., 2021; Shah et al., 2023); MSCCLang compiles a collective DSL for a GPU runtime (Cowan et al., 2023). MSCCL++ exposes GPU-initiated data movement and synchronization primitives, a collective DSL, and an NCCL-compatible API. Its PortChannel delegates transfer submission to a CPU proxy, while the GPU program issues communication requests and executes the associated waits (Hwang et al., 2026; Microsoft, 2026). HostCCL executes existing ring schedules with CPU-managed dependencies, keeping inter-rank readiness waits outside the GPU worker.

7 Conclusion

HostCCL decouples collective progress from GPU execution. A CPU executor advances tile dependencies, copy engines move payloads, and a configurable GPU worker executes ready reductions. The runtime configures transfer concurrency independently of reduction parallelism. On four and eight PCIe-connected L40S GPUs, AllGather and AllToAll launch no steady-state GPU kernels. ReduceScatter and AllReduce use a one-SM budget per GPU, with a 50–75% smaller SM budget and up to 23.3% higher throughput than default NCCL for the tested large FP32 reductions. In a controlled four-GPU AllGather–GEMM experiment, HostCCL also reduces interference with concurrent computation and shortens joint completion time relative to default NCCL.

AI use statement

Generative AI tools assisted with manuscript polishing, reference formatting, and data-analysis scripts. The authors designed the study and verified all text, figures, code, data, and conclusions.

Reproducibility statement

Section 5.1 and Appendix A document the software versions, CPU placement, timing protocol, and correctness tests. Appendices F.1–F.7 specify the implementation parameters and synchronization contracts.

References

Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. Synthesizing optimal collective algorithms. In *26th ACM SIGPLAN Symposium*

on Principles and Practice of Parallel Programming (PPoPP), pp. 62–75. ACM, 2021. doi: 10.1145/3437801.3441620. URL https://ppopp21.sigplan.org/details/PPoPP-2021-main-conference/43/Synthesizing-Optimal-Collective-Algorithms.

Meghan Cowan, Saeed Maleki, Madanlal Musuvathi, Olli Saarikivi, and Yifan Xiong. MSCCLang: Microsoft collective communication language. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Volume 2*, pp. 502–514. ACM, 2023. doi: 10.1145/3575693.3575724. URL https://doi.org/10.1145/3575693.3575724.

Changho Hwang, Peng Cheng, Roshan Dathathri, Abhinav Jangda, Saeed Maleki, Madan Musuvathi, Olli Saarikivi, Aashaka Shah, Ziyue Yang, Binyang Li, Caio Rocha, Qinghua Zhou, Mahdieh Ghazimirsaeed, Sreevatsa Anantharamu, and Jithin Jose. MSCCL++: Rethinking GPU communication abstractions for AI inference. In *31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Volume 2*, pp. 1201–1215. ACM, 2026. doi: 10.1145/3779212.3790188. URL https://doi.org/10.1145/3779212.3790188.

Sylvain Jeaugey, John Bachan, Pak Markthub, Zhenhao He, Sirshak Das, and Farshad Ghodsian. Fusing communication and compute with new device API and copy engine collectives in NVIDIA NCCL 2.28. NVIDIA Technical Blog, https://developer.nvidia.com/blog/fusing-communication-and-compute-with-new-device-api-and-copy-engine-collectives-in-nvidia-nccl-2-28/, 2025. Accessed September 26, 2026.

Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 463–479. USENIX Association, 2020. URL https://www.usenix.org/conference/osdi20/presentation/jiang.

Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. Evaluating modern GPU interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110, 2020a. doi: 10.1109/TPDS.2019.2928289. URL https://ieeexplore.ieee.org/document/8763922.

Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. PyTorch distributed: Experiences on accelerating data parallel training. *Proceedings of the VLDB Endowment*, 13(12):3005–3018, 2020b. doi: 10.14778/3415478.3415530. URL https://www.vldb.org/pvldb/vol13/p3005-li.pdf.

Tongrui Liu, Chenyang Hei, Fuliang Li, Chengxi Gao, Jiamin Cao, Tianshu Wang, Ennan Zhai, and Xingwei Wang. ResCCL: Resource-efficient scheduling for collective communication. In *ACM SIGCOMM Conference*, pp. 55–70. ACM, 2025. doi: 10.1145/3718958.3750514. URL https://doi.org/10.1145/3718958.3750514.

Microsoft. MSCCL++: A GPU-driven communication stack for scalable AI applications. https://github.com/microsoft/mscclpp, 2026. Accessed September 25, 2026.

NVIDIA. *CUDA Programming Guide*. NVIDIA, 13.3.0 edition, 2026a. URL https://docs.nvidia.com/cuda/archive/13.3.0/cuda-programming-guide/index.html. Sections 4.15 (Inter-Process Communication) and 5.7 (CUDA C++ Memory Model); accessed September 25, 2026.

NVIDIA. *CUDA Driver API: Stream Memory Operations*. NVIDIA, 13.3.0 edition, 2026b. URL https://docs.nvidia.com/cuda/archive/13.3.0/cuda-driver-api/group__CUDA__MEMOP.html. Accessed September 25, 2026.

NVIDIA. GDRCopy: A fast GPU memory copy library based on NVIDIA GPUDirect RDMA technology. https://github.com/NVIDIA/gdrcopy, 2026c. Accessed September 25, 2026.

NVIDIA. NVIDIA collective communications library (NCCL). https://github.com/NVIDIA/nccl, 2026d. Accessed September 25, 2026.

NVIDIA. NCCL environment variables and CTA budgets. https://docs.nvidia.com/deeplearning/nccl/archives/nccl_2312/user-guide/docs/env.html, 2026e. NCCL 2.31.2 documentation; accessed September 25, 2026.

NVIDIA. NCCL collective buffer registration implementation. https://github.com/NVIDIA/nccl/blob/v2.31.2-1/src/register/coll_reg.cc, 2026f. Version 2.31.2-1; accessed September 25, 2026.

NVIDIA. NCCL 2.31.2: Ring ReduceScatter implementation. https://github.com/NVIDIA/nccl/blob/v2.31.2-1/src/device/reduce_scatter.h, 2026g. Tag v2.31.2-1; accessed September 20, 2026.

NVIDIA. NCCL 2.31.2: Simple protocol device primitives. https://github.com/NVIDIA/nccl/blob/v2.31.2-1/src/device/prims_simple.h, 2026h. Tag v2.31.2-1; accessed September 20, 2026.

NVIDIA. Performance reported by NCCL tests. https://github.com/NVIDIA/nccl-tests/blob/master/doc/PERFORMANCE.md, 2026i. Accessed September 20, 2026.

NVIDIA. NCCL user buffer registration and zero-CTA optimization. https://docs.nvidia.com/deeplearning/nccl/archives/nccl_2312/user-guide/docs/usage/bufferreg.html, 2026j. NCCL 2.31.2 documentation; accessed September 25, 2026.

Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009. doi: 10.1016/j.jpdc.2008.09.002. URL https://www.sciencedirect.com/science/article/pii/S0743731508001767.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory optimizations toward training trillion parameter models. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1–16. IEEE, 2020. doi: 10.1109/SC41405.2020.00024. URL https://doi.org/10.1109/SC41405.2020.00024.

Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. TACCL: Guiding collective algorithm synthesis using communication sketches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 593–612. USENIX Association, 2023. URL https://www.usenix.org/conference/nsdi23/presentation/shah.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019. doi: 10.48550/arXiv.1909.08053. URL https://arxiv.org/abs/1909.08053.

Guanbin Xu, Zhihao Le, Yinhe Chen, Zhiqi Lin, Zewen Jin, Youshan Miao, and Cheng Li. AutoCCL: Automated collective communication tuning for accelerating distributed and parallel DNN training. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 667–683. USENIX Association, 2025. URL https://www.usenix.org/conference/nsdi25/presentation/xu-guanbin.

Mingjun Zhang, Xiaohe Hu, Menghao Zhang, Ziteng Chen, Yanmin Jia, Yan Zhang, Da Liu, Qing Chen, Fangzheng Jiao, Jun Chen, He Liu, Aohan Zeng, Shuaixing Duan, Ruya Gu, Yang Jing, Bowen Han, Wei Chen, Wenqi Xie, Jinlong Hou, Yuan Cheng, Hongzhou Zhang, Bohua Xu, Mingwei Xu, and Chunming Hu. An efficient, reliable and observable collective communication library in large-scale GPU training clusters. *arXiv preprint arXiv:2510.00991*, 2025. doi: 10.48550/arXiv.2510.00991. URL https://arxiv.org/abs/2510.00991v2. Version 2, revised June 1, 2026.

Yang Zhou, Zhongjie Chen, Ziming Mao, ChonLam Lao, Shuo Yang, Pravein Govindan Kannan, Xizhi Zhang, Jiaqi Gao, Yilong Zhao, Yongji Wu, Kaichao You, Fengyuan Ren, Zhiying Xu, Costin Raiciu, and Ion Stoica. UCCL-Tran: An extensible software transport layer for GPU networking. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 1143–1166. USENIX Association, 2026. URL https://www.usenix.org/conference/osdi26/presentation/zhou-yang.

A Experimental Details

A.1 Build and placement

All experiments use CUDA 13.3 and native NCCL 2.31.2. HostCCL is built for SM 89 with a reduction instruction-level parallelism (ILP) factor of four and the Tensor Memory Accelerator (TMA) path disabled. The primary benchmark is built with `-O3 -std=c++17 -arch=sm_89` against native NCCL headers.

The dual-socket Intel Xeon Platinum 8350C server has 64 physical CPU cores and eight L40S GPUs. GPUs 0–3 attach to one NUMA domain and GPUs 4–7 to the other. Four-GPU runs use GPUs 0–3; eight-GPU runs span both domains. For local rank r, the main CPU mask includes logical CPU IDs $8r$ through $8r + 7$ and their simultaneous multithreading (SMT) siblings, IDs $8r + 64$ through $8r + 71$. The launcher applies this mask before library initialization. The masks are disjoint across ranks and identical for both libraries in each paired configuration. They specify allowed cores, rather than exclusive reservations or measured CPU usage; summed CPU time is reported separately.

HostCCL uses a single FIFO and one IPC stream per peer. Reducing schedules use the ring algorithm and a fixed worker SM budget. Movement operations do not launch that worker. Transfer-notification integration and applicable direct-final-output reductions remain enabled, with one tile per notification wait. Calls have ordinary per-call group boundaries. The measurements do not use multi-call tensor fusion. Native default has no CTA, algorithm, protocol, or efficiency-policy overrides; fixed-SM-budget runs and the efficiency policy are labeled separately. The primary benchmark uses ordinary CUDA allocations without explicit native buffer/window registration. The registration controls below vary allocation and registration while keeping the library builds unchanged.

A.2 Metrics, timing, and verification

For N ranks and slowest-rank wall time t, our bus-bandwidth metric is $S(N-1)/(Nt)$ for AllGather, ReduceScatter, and AllToAll, and $2S(N-1)/(Nt)$ for AllReduce, following nccl-tests conventions (NVIDIA, 2026i). Here S is full per-rank output for AllGather, full input for the reducing collectives, and full send volume for AllToAll. We compute bandwidth from the slowest-rank time without summing independently measured rank rates. This normalized metric does not measure physical link utilization. Bandwidth uses decimal GB/s, while message sizes use binary units. Below, AG, RS, and AR denote AllGather, ReduceScatter, and AllReduce; numeric suffixes give the size in MiB.

The full size set is 4 KiB, 64 KiB, 1 MiB, 4 MiB, 16 MiB, 64 MiB, 128 MiB, and 256 MiB. The size sweep synchronizes every call, using 2,048 timed calls through 4 MiB, 128 at 16 MiB, and 40 at larger sizes. The SM-budget sweep synchronizes every four calls, using 2,048 calls for 4 MiB ReduceScatter and 40 for the large-message cases. Initialization, four changing-input stream-consumer checks, and five warmups precede timing. Each configuration has three independent process launches; the figures report their median and full range. Summed CPU-ms/call includes timed user plus system CPU time across all participating ranks.

Each correctness epoch changes the input and checks the result through a consumer ordered on the application stream. Additional correctness tests cover worker generation and retirement and device batches. Compute Sanitizer's `memcheck` reports no errors in 60 guarded FP32 small-slice cases covering copy and reduction helpers, zero-length inputs, and vector remainders.

A.3 Native buffer registration

Figure 5 compares native NCCL with ordinary `cudaMalloc`, `ncclMemAlloc` without registration, and `ncclMemAlloc` with `ncclCommRegister`. In the registered configuration, all ranks register both buffers before verification and warmup, and deregister them after execution. Comparing the latter two configurations separates registration from the allocator change. The control covers AG256, RS128, and AR256 on four/eight GPUs: 54 launches, with three repetitions per configuration, 40 timed calls, per-call synchronization, and the CPU masks above. CTA, algorithm, and protocol selection remain automatic.

The largest median change from registration is four-GPU AG256: 21.49 to 22.01 GB/s (2.4%). All other tested points change by less than 1%. Separate diagnostics confirm IPC registration for

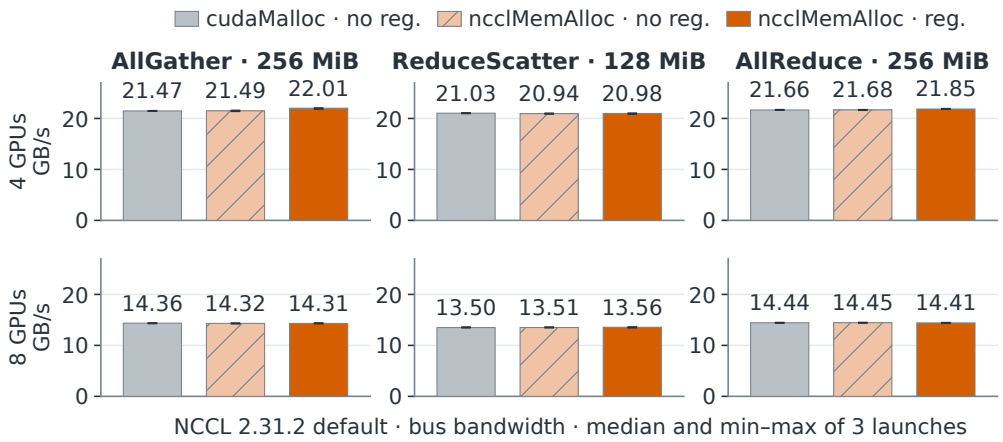

Figure 5: Native NCCL allocator and registration controls. Bars show bus bandwidth with default tuning; medians and min–max of three launches, synchronizing every call. "reg." denotes an explicit buffer-registration request, whose use depends on the selected path.

four-GPU Ring/Simple AllGather and out-of-place AllReduce. Registration requests do not guarantee an optimized path: NCCL 2.31.2's ordinary IPC registration branch skips Ring ReduceScatter (NVIDIA, 2026f). Eight-GPU diagnostics report that the all-direct-P2P prerequisite is absent, so registration does not activate that path. These controls concern general buffer registration, not the topology-dependent zero-CTA path (NVIDIA, 2026j).

A separate one-SM-budget control, configured with one CTA, matches the eight-GPU AR256 budget protocol with synchronization every four calls. Three launches per allocation/ registration configuration give medians of 11.84, 11.91, and 11.91 GB/s, respectively. The path diagnostics above use default CTA selection and per-call synchronization.

A.4 Observed GPU execution grids

The resource traces cover four timed calls per configuration after initialization, verification, and warmup, with per-call synchronization matching the main size sweep. The 128 configurations cover two rank counts, four collectives, eight sizes, and two libraries. Every participating rank is profiled. From kernel grids and execution intervals, we compute the peak sum of overlapping communication-kernel grid blocks in the timed interval. Bandwidth comes from separate, unprofiled runs.

HostCCL's AG/A2A captures contain no steady-state GPU kernel launches; all RS/AR captures have a peak grid sum of one. Native AG/RS/AR use one block at 4 KiB and four/two blocks on four/eight GPUs at every other measured size. Native AllToAll uses four/two at all sizes. Separate budget-control captures verify one-, 32-, and 64-block representative settings on every rank. An SM budget of B is configured with B communication blocks per GPU. The reported SM budget comes from these communication-kernel grid counts, not sampled SM utilization or a count of distinct resident SMs; it does not imply exclusive SM reservation. Budget-sweep results use requested SM budgets, set through the communication-block count.

B Controlled Compute–Communication Overlap

The same-process GEMM/CCL experiment uses the same HostCCL runtime as the collective benchmarks. Each of four ranks initializes one communicator and two equal-priority nonblocking streams. Separate buffers hold the collective data and GEMM matrices. FP32 AllGather uses a full 256 MiB output per rank; FP32 Sum ReduceScatter uses a full 128 MiB input. GEMM uses 4096×4096 matrices, FP16 inputs, FP32 accumulation/output, and cuBLAS tensor operations with $\beta = 0$. An independent native compute-only calibration measures 579.59 μs per GEMM. Rounding the predefined

9/4.5 ms work targets to integer counts selects 16/eight GEMMs for AG/RS, unchanged across all configurations.

Each configuration runs compute-only, CCL-only, sequential execution, and concurrent execution under both submission orders. Sequential mode orders GEMM after CCL with an event dependency. Concurrent mode enqueues both streams without a cross-stream dependency; every iteration waits for both to finish. Host wall time includes submission, event observation, and final device synchronization. CUDA events measure the interval between start and stop events on each stream. In sequential mode, the compute start event follows the CCL dependency wait, so that wait is excluded from the compute-stream duration. Compute-only mode retains the initialized communicator and runtime threads. All kernel and cuBLAS shapes are warmed before asynchronous CCL execution.

The 90 launches cover two collectives, three library configurations, five mode/order choices, and three repetitions. Each launch uses five warmups and 40 timed iterations. The second repetition reverses library order. Four changing-input epochs precede timing and a fifth follows it; all active collective and GEMM outputs are checked. The full GEMM uses an analytic reference; an additional small, nonseparable GEMM is checked against a CPU reference. All ranks use the CPU allowances described above. Slowdowns divide concurrent stream durations by the corresponding standalone duration for the same collective, library, and repetition. We report medians and full ranges of these ratios. Each duration metric is maximized across ranks independently.

In Figure 4, AG joint completion improves under HostCCL for both submission orders, whereas the RS communication slowdown offsets its smaller compute slowdown. Host CPU cost also differs: AG overlap consumes about 242 CPU-ms/iteration summed across four ranks, versus 59–60 for default NCCL; RS consumes about 216 versus 31. This includes event observation and runtime polling. The fixed GEMM workload measures how concurrent collective communication affects compute latency and joint completion time.

Profiling all four ranks across the 12 concurrent configurations confirms temporal overlap between GEMM and collective communication. HostCCL AG uses CE transfers and no communication kernel; RS has a peak communication grid sum of one, corresponding to a one-SM budget. Default NCCL's four communication blocks give a four-SM budget; the one-SM configuration uses one communication block for both operations. The GEMM and NCCL communication kernels execute concurrently. The performance figure uses unprofiled measurements.

C Tile Progression Diagnostics

We instrument HostCCL to distinguish CPU completion observation from transfer delivery. Four ranks execute four FP32 ReduceScatter calls with a one-SM worker budget and per-call synchronization. RS4 uses 1 MiB tiles and RS128 uses 2 MiB tiles. CUDA/NVTX traces, host dependency records, and device FIFO timestamps identify every reduction and its dependent forward. Sixteen calibration brackets per rank map the device timer into the trace clock; maximum offset uncertainty is below 0.5 μs. The traces contain 32/512 dependent forwards for RS4/RS128, including the first captured call. Instrumentation changes timing, so these intervals do not replace unprofiled performance samples.

For RS128, the CPU observes a reduction's device completion marker after a median 1.36 μs; completing its DAG node takes another median 0.84 μs. Figure 6 separates this completion observation from subsequent transport delays. The median interval from the device completion marker to the forward copy's start is 1,109 μs, with a median 850 μs between transport binding and CUDA issue. The outgoing stream executes other CE copies for a median 98% of that interval. At binding, 492 of 512 forwards encounter a full four-copy issue window. The occupied issue window and busy CE stream point to transport backlog as the main source of the long bind-to-issue interval; the exact request-queue acceptance instant is not instrumented.

After CUDA API return, the new copy waits a median 239 μs to start. Its same-stream predecessor accounts for a median 237 μs of remaining work, and the copy starts a median 1.50 μs after that predecessor finishes.

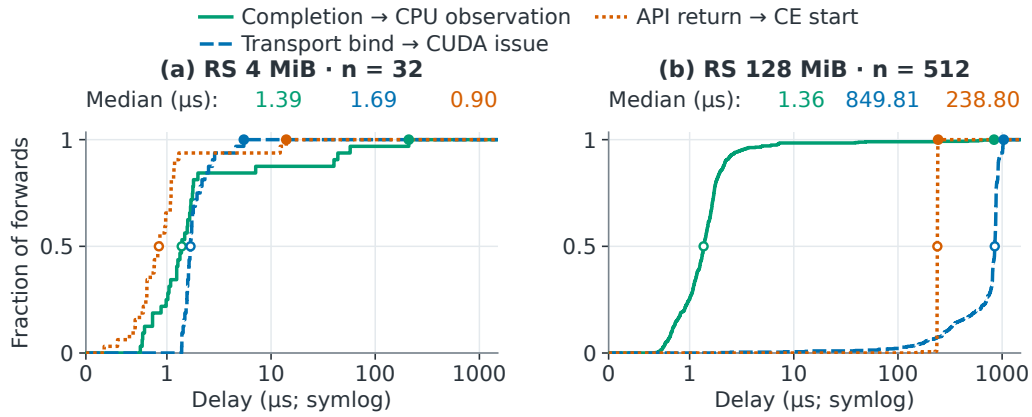

Figure 6: Distributions of three per-forward intervals in instrumented four-GPU ReduceScatter, covering all 32/512 forwards across four calls. "Completion" refers to the GPU reduction's completion marker. The horizontal axis is linear near zero and logarithmic beyond 1 μs. Each interval has its own median; the medians are not additive.

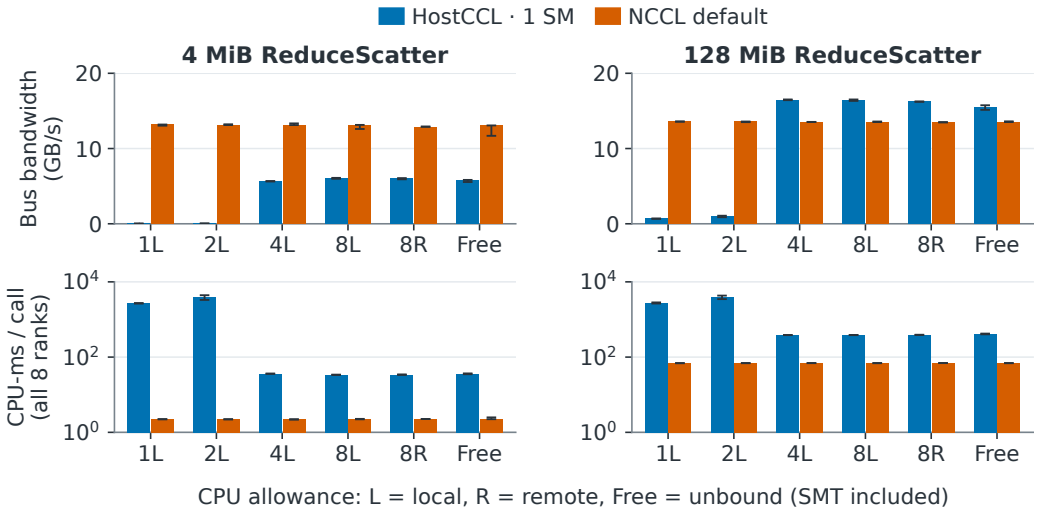

Figure 7: Eight-GPU host-provisioning control. L/R denote local/remote physical-core allowances, including SMT siblings; Free leaves placement unbound. Both libraries receive the same initial mask. Top: collective bandwidth. Bottom: summed CPU cost on a logarithmic axis. Points show medians and min–max over three independent launches, synchronizing every four calls.

D Host Provisioning

Figure 7 varies the initial CPU allowance while retaining a one-SM HostCCL worker budget and native default. Local and remote masks are disjoint across ranks and include SMT siblings. Unbound placement leaves the initial mask unrestricted. The allowed cores are not reserved exclusively; we separately measure the accumulated CPU time of participating processes.

For 128 MiB ReduceScatter, four local cores per rank provide 16.45 GB/s, versus 16.37 with eight. One/two cores give 0.69/0.96 GB/s, while summed CPU cost rises to 2,703/3,882 CPU-ms/call versus 386 with four. The smaller allowances prolong each call enough to increase its accumulated CPU time as well. The required allowance also depends on the workload: at 4 MiB, four cores provide 5.67 GB/s versus 6.08 with eight. The main curves use the same eight-core allowance for every size.

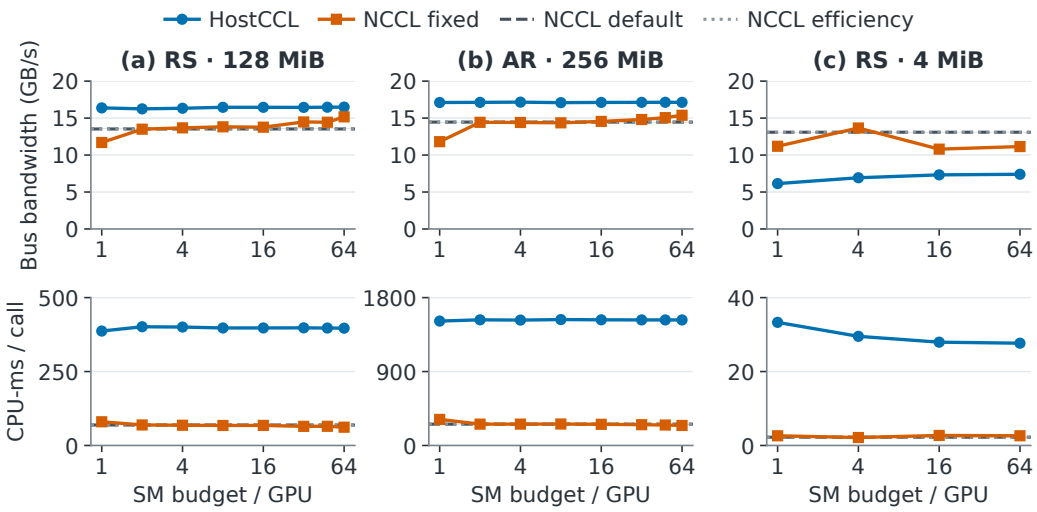

Figure 8: Full eight-L40S collective SM-budget sweep. Top: bus bandwidth. Bottom: CPU time summed across eight ranks. Columns show 128 MiB ReduceScatter, 256 MiB AllReduce, and 4 MiB ReduceScatter. Fixed-budget points show requested SM budgets, set through communication-block counts; native default/efficiency are separate references. Three-launch medians and min–max; synchronization every four calls.

With eight allowed cores, remote placement provides 16.24 GB/s at 128 MiB, compared with 16.37 locally. Unrestricted initial placement provides 15.50 GB/s. Native RS128 medians stay between 13.51 and 13.60 GB/s. The control uses 128 timed calls at 4 MiB and 40 at 128 MiB; calls synchronize every four submissions. Native timing windows for the smaller size are shorter than in the primary size sweep. The measurements capture the combined effects of scheduling and NUMA traffic.

E SM Budgets and Worker Capacity

E.1 Full collective SM-budget sweep

Figure 8 reports all 138 SM-budget runs: 46 configurations with three independent launches each. It includes 4 MiB ReduceScatter and summed CPU cost, extending the main SM-budget panels. Calls synchronize every four submissions. Default and efficiency policies leave the communication-block count to NCCL; they are plotted separately from fixed-SM-budget configurations.

With a one-SM worker budget, 128 MiB ReduceScatter uses 386.95 CPU-ms/call across eight ranks, versus 69.24 for native default. For 256 MiB AllReduce, the corresponding costs are 1,514.55 and 259.38 CPU-ms/call. HostCCL uses a smaller SM budget and more CPU time for these calls; the two metrics measure different resources.

For a common target rate T, define the minimum tested SM budget as $B_{\min}(T) = \min\{B \in \mathcal{B} : \widehat{W}(B) \geq T\}$, where $\mathcal{B}$ contains the fixed SM budgets in the sweep and $\widehat{W}$ is median collective bus bandwidth. An independent NCCL pilot compares default tuning, the efficiency policy, and a 64-SM budget; its highest rates are 15.15 GB/s for RS128 and 15.35 GB/s for AR256, both at a 64-SM budget. These rates set common targets for the subsequent budget comparison. At 90% of the reference rates, the minimum tested SM budgets are one for HostCCL and four/two for NCCL on RS/AR. At 95%, they are one for HostCCL and 32 for NCCL on both operations. These minima describe the tested budgets at each target; NCCL's automatically selected default remains the separate reference in Figure 8.

E.2 READY-WORK SERVICE MEASUREMENT

Each of eight GPUs receives a fixed 112 MiB reduction payload from two FP32 input arrays, a 224 MiB input working set exceeding the reported 96 MiB L2 capacity. The worker reads inputs with the external-operand load policy described in Appendix F.5 and reduces in place. Tasks are supplied before device timing, so no collective transport delivers inputs during the measured interval. Five warmups precede 30 timed rounds; all 35 changing-input epochs are verified outside timing. A point is payload bytes divided by the slowest rank's mean CUDA-event time, summarized over three independent launches. Figure 3 uses 2 MiB tasks and worker SM budgets from one to 64. Worker payload rate measures service capacity with ready inputs; collective bus bandwidth includes dependency progress and transport.

F RUNTIME IMPLEMENTATION DETAILS

The evaluated PCIe configuration (Section 3) uses ring reducing schedules, one worker FIFO, CE transfers through CUDA IPC mappings, and CPU-owned peer readiness. AllGather and AllToAll require no reduction worker; ReduceScatter and AllReduce use a one-SM worker budget (one block) in the main comparison.

F.1 REUSABLE PLANS AND TILE DEPENDENCIES

An algorithm builder emits a rank-local graph of transfers, receives, and receive–reduce operations. Each operation specifies buffer roles, byte ranges, peers, transfer-pair identifiers, and local predecessors. The builder declares temporary storage explicitly. Lowering partitions these operations into tiles and inserts arrival waits and notifications. Tile dependencies require equal byte counts at an edge's endpoints; lowering rejects unequal-length edges.

The cached plan holds backend command templates, successor lists in compressed sparse row form, initial dependency counts, and initially ready operations. Each run has its own buffer bindings, invocation epoch, ready queues, and completion state. When a producer completes, one segment record can represent its single-predecessor successors by referring to their precompiled fan-out. Dispatch expands this record; multi-predecessor joins retain atomic dependency counters. One segment record replaces separate readiness records while preserving every dependency edge; it does not combine operations into a GPU kernel.

Transport tile size and worker SM budget, set through the block count, are independent parameters. The default tiling rule uses a 1 MiB floor, rounds the computed size to 32 bytes, and targets 256 tiles for messages up to 64 MiB and 64 for larger messages. These targets determine tile size; the full multi-hop graph can contain more operations. An individual algorithm shard can be smaller than the floor.

Composition and direct output. In ring AllReduce, the builder carries each shard's latest producer from ReduceScatter into AllGather. Lowering retains tile-level dependencies rather than inserting a whole-phase barrier. Eligible final Sum reductions read the incoming partial and local input and store directly to the final destination. For the evaluated FP32 Sum paths, the final ReduceScatter step produces the shard owned by the local rank. Standalone ReduceScatter writes each tile at its offset within the rank's output; in-place AllReduce writes at the owned shard's offset in the full output, avoiding a separate CE copy from scratch. Out-of-place ring AllReduce already reduces into output. Successors reference the actual result destination, even when it differs from the incoming partial's storage. Both phase composition and direct output apply within one collective invocation.

F.2 HOST PROGRESS AND COMPLETION OWNERSHIP

The executor's enqueue thread dispatches ready operations; separate threads drain device, transport, and signal completions. The executor rebuilds its snapshot of active runs when submissions change, avoiding a registry copy on every poll. Dispatch binds command templates to each run's buffers and protocol identities.

Before exposing a command to a backend, the executor reserves its index, records the owning run and operation, and increments its in-flight reference count. Ownership slots transition through empty,

writing, ready, and reading states with full index generations. A completion consumer copies the recorded owner before freeing the slot. Rejected commands roll back their references. An immediate completion can therefore recover its owner.

Backends accept a prefix of each batch; the executor returns the unaccepted suffix to its runs. Dispatch continues to later peers when a transport peer is blocked. Signal waits have a separate in-flight cap from outgoing signals, so pending receives do not exhaust the outgoing-signal budget. Draining a run's final operation completes the run directly. Encoded backend request identifiers avoid a separate context-map lookup for transport and signal completions.

F.3 CE TRANSPORT AND INTEGRATED NOTIFICATION

Peer transfers use CUDA IPC mappings and asynchronous device-to-device copies. The adapter maintains per-peer data and control queues, copy streams, event pools, and bounded in-flight windows. A busy peer cannot consume another peer's event pool. Control notifications progress independently of copy-window occupancy. The evaluated configuration uses one copy stream per peer. Matching sequences are assigned after enqueue acceptance, so rejection cannot introduce a sequence gap.

A `PutSignal` combines movement and notification in one adapter request. After observing copy completion, the adapter publishes local transfer completion and writes the peer's readiness tag. A full peer signal ring defers that notification without blocking other peers. The receiver's CPU wait becomes ready when the tag arrives. This fusion combines backend commands without an SM copy kernel. Local transfer completion and remote readiness remain separate events.

For out-of-place AllGather and AllToAll, the entry layer copies the rank's own slice with a CE operation on the user stream before recording input readiness. A plan flag suppresses a duplicate local copy. Local puts handled by the device backend also use a copy stream and completion event. The host manages the remaining movement and notification, so steady-state calls without reductions launch no worker kernel. The initial empty worker warmup precedes these calls. AllToAll rotates the planned destination order by rank to distribute early submissions across peers.

F.4 ASYMMETRIC CPU–GPU QUEUES

C2D publication. A CPU-to-device FIFO has one CPU producer and one cooperating GPU grid as its logical consumer. Each entry contains a 16-byte descriptor; operand addresses, length, and flags occupy a separate 96-byte argument slot. Both use host-writable GDR mappings of GPU memory (NVIDIA, 2026c). The CPU-owned head is in pinned host memory; the GPU-owned tail is in GPU memory with a host-readable GDR mapping. Monotonic head and tail sequences select slots modulo capacity. Tensor data remain outside these control structures.

The producer writes arguments and descriptors, flushes host write-combining buffers, and release-publishes the head. One publication may expose several ready tasks. The first task is published before a generation launches; an open generation stages at most eight ready tasks before publication. After the producer's flush and release store, block 0 reads the head with a system-scope acquire load (NVIDIA, 2026a) and release-publishes a device-resident hint. Other blocks acquire this hint before loading descriptors and arguments.

D2C completion. Reductions retire in FIFO order after every cooperating block finishes. Output writes are ordered before tail publication. The CPU reads the tail through its GDR mapping and retires pending sequences below that frontier. Pending records map sequences to backend command indices and argument slots. A single D2C completion counter replaces a queue of per-task messages. One CPU observation can reclaim several tasks, although the GPU publishes progress at every task completion. The worker may continue running after a task completes.

Capacity and reuse. With capacity C, head h, observed tail t, and a proposed batch of n entries, admission requires $h - t + n < C$. The CPU reserves one entry for `EndBatch`, so it can request retirement without waiting for a free task slot. A stale tail only reduces available capacity. Descriptor reuse follows the completion frontier; argument slots return to their per-FIFO pools only when their pending records retire.

F.5 COOPERATIVE ARITHMETIC AND FINITE GENERATIONS

Task execution. Every block assigned to a FIFO consumes the same task and reduces a disjoint data partition. Blocks do not take independent tasks from that FIFO. Block 0 refreshes the host head; other blocks read its device-resident hint. They drain the visible prefix before polling for more work. For B blocks and task sequence s, a monotonic arrival counter reaches $B(s+1)$ after all blocks finish. The last arrival performs the required output ordering and publishes the tail. Other blocks observe both tail publication and the arrival threshold before advancing, avoiding reset races and duplicate task execution. The one-block path omits inter-block atomics and polling while retaining local barriers and publication ordering.

Each block's leader loads the descriptor and arguments into shared memory. Vector partitions use 16-byte boundaries with scalar handling of misalignment and remainders. The worker uses typed Sum routines and instruction-level parallelism, with an ILP factor of four in the evaluated build. Separate compilation limits the dispatch loop's register footprint, and an aligned FP32 Sum entry bypasses generic forwarding and datatype dispatch.

External operands rewritten by CE or application kernels use vector or scalar `__ldcv` loads. Cache policy does not replace synchronization: producer completion and readiness must precede reads, and result visibility must precede CPU-observed completion. Repeated changing-input tests exercise this visibility requirement across tasks and generations.

Retirement. The CPU decides when to close a generation. A dispatch pass seals open generations with `EndBatch` if it has no new device submissions and no ready device work deferred by backpressure. This ordered marker cannot be withdrawn: work accepted after sealing belongs to a successor generation. All blocks consume the marker through the task protocol and return. The successor launches on the same worker stream, behind its predecessor. Block 0 anchors its arrival counter to the current device tail and publishes a generation identifier; no host counter reset or per-launch host synchronization is required. Block count is fixed within a generation. Initialization warms an empty sealed generation; movement-only calls create none during steady state.

F.6 STREAM ADMISSION, BUFFERS, AND FAILURE

For each run, the runtime records input readiness and installs a stream completion ticket using a 32-bit CUDA stream memory wait (NVIDIA, 2026b). The 256-entry ticket pool places tickets on separate cache lines. The runtime reuses a ticket only after an event following its previous gate completes, so a later completion cannot release an earlier run. Operations on a stream retain submission order; independent streams can become ready separately.

Admission claims scratch lanes, selecting among four lanes per scratch size by invocation epoch. A run waits if another run owns its lane. The runtime checks all required scratch resources before claiming any, avoiding partial acquisition. Receiver-admission notifications are sent only after the stream and scratch gates permit execution. These notifications protect both initial and subsequent remote writes.

Registration tracks allocation extent and identity, plus tensor offset. Reusing a virtual address for a new allocation changes its registration identifier and invalidates dependent preparation entries. The cold path publishes local storage and imports the remote destinations used by the plan before installing stream gates. This prevents a lazy mapping from waiting behind a gate whose transfer needs that mapping. Registrations and IPC mappings are reused across offsets of the same allocation.

In-place AllReduce protects still-needed input with accumulation scratch; in-place AllToAll stages sends before incoming writes can overwrite them. The collective interface assumes corresponding allocation histories and collective issue order across ranks, and buffers must remain live while peers can access them. Run epochs distinguish reused protocol tags within their encoding bounds. A sticky executor error rejects new submissions; reclamation waits for dispatch retirement and outstanding completions. Outputs from cancelled runs are invalid.

F.7 CPU THREAD AND MEMORY PLACEMENT

IPC send and receive progress threads run alongside the enqueue thread and the device, transport, and signal completion threads. Delayed readiness observation postpones the next tile operation even when its data have already arrived. With no active runs, the enqueue thread sleeps on a condition variable; with pending waits, the signal completion thread uses short pause bursts.

The launcher sets each rank's affinity mask before library initialization. New runtime threads inherit that mask; the runtime does not assign each thread a dedicated core. Main experiments allow eight GPU-local physical cores and their SMT siblings per rank, with disjoint rank masks and identical initial core allowances for both libraries. Thread-affinity measurements confirm inheritance of these masks. CPU use is measured as accumulated process time (Appendix D).

The FIFO constructor also requests a preferred allocation policy for the GPU's NUMA node before allocating control storage. This memory policy does not pin a thread or establish the placement of every page. CPU affinity controls thread placement, NUMA policy guides control-storage allocation, and the worker block count sets the SM budget through GPU launch geometry.