# COLLECTIVE ALGEBRA ABOVE A CLOSED LIBRARY: A RESEARCH-LOOP AGENT FINDS THE REWRITES A SEARCH MISSES

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Collective-communication libraries on closed accelerator stacks such as AWS Trainium expose a fixed set of primitives and provide little control over how they are scheduled. OverlayCCL searches above this library layer: a language model uses an on-device-calibrated simulator to generate, implement, and evaluate different collective-communication strategies, then refines the most promising ones. However, enumerating structural rewrites alone misses important optimizations. For example, OverlayCCL may miss cases where a value that appears to depend on the current computation is actually a constant determined by the routing pattern, allowing several communication operations to be replaced by a single one. It may also miss opportunities to replace a Python loop that repeatedly prepares data for an unavoidable collective operation with a vectorized operation that performs the same preparation in one step.

We therefore introduce SorcarCCL, a research agent that extends OverlayCCL by proposing and evaluating such optimizations using the same simulator and hardware constraints. SorcarCCL compares ideas before implementation, retains validated improvements, records failures, and runs a separate adversarial pass. On the eight collective problems evaluated by OverlayCCL, SorcarCCL matches its results without regression. On 40 more complex production collective problems, we search every problem with nine seeds. Across all 40 problems, the median-seed rewrites run on 224 Trainium cores at $1.12\times$ to $4.37\times$ the performance of OverlayCCL's solutions, with every problem improved by SorcarCCL. Integrated into end-to-end training of 9.7-billion-parameter GPT- and Llama-style transformers on 224 cores, SorcarCCL's rewrites further reduce steady-state step time by $2.13\times$ and $2.15\times$ relative to OverlayCCL's solutions.

## 1 INTRODUCTION

Training a large model spreads its computation across hundreds of accelerators, making collective communication a substantial part of every training step. On NVIDIA clusters, collective libraries expose enough control to synthesize and install schedules below the library interface (Cai et al., 2021; Cowan et al., 2023; Shah et al., 2023). AWS Trainium imposes a different boundary: Neuron, the closed compiler-and-runtime stack that AWS ships for Trainium, exposes collectives only as `xm.all_reduce`, `xm.all_gather`, `xm.reduce_scatter`, `xm.collective_permute`, and `xm.all_to_all` through PyTorch/XLA, with no schedule-level intermediate representation and compiled implementations hidden from users (Amazon Web Services, 2026). Ke et al. (2026) report similar constraints for Google TPU and Meta MTIA.

OverlayCCL (Ke et al., 2026) addresses this constraint by searching above the collective-library boundary: an LLM agent builds a simulator calibrated to the target cluster, generates different collective strategies, implements and evaluates them, and refines promising candidates, each of which must compile and run at training scale before deployment. This is effective when the main opportunity is to choose a better primitive or data layout—for example, in the AllToAllV that routes tokens to experts in a mixture-of-experts layer (Lepikhin et al., 2021), it can discover alternatives such as pack-then-gather versus gather-then-reduce-scatter. On the AWS production baseline, OverlayCCL reports $1.40\times$ on OLMoE-10B and $3.24\times$ on Llama-7B, outperforming both a coding-agent baseline



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.

such as Claude Code (Anthropic, 2025) and multi-island evolutionary search in the style of FunSearch, AlphaEvolve, and ShinkaEvolve (Romera-Paredes et al., 2024; Novikov et al., 2025; Lange et al., 2025).

However, choosing and configuring collectives is not the only source of optimization; some opportunities require recognizing that the computation itself can be rewritten. Consider a two-round mixture-of-experts reduction $y = \mathrm{AR}(x_1)/c_1 + \mathrm{AR}(x_2)/c_2$. Each $c_i$ is a coverage count obtained from the routing table and therefore appears to depend on the current computation. But the routing table is fixed across steps, so the $c_i$ are actually step-invariant constants. This makes it possible to rewrite the computation as $y = \mathrm{AR}(x_1/c_1 + x_2/c_2)$, reducing two collective dispatches to one. OverlayCCL can rearrange and re-bucket the existing all-reduces, but its structural search does not discover this cross-operation identity because it does not reason about the step-invariance of the values involved. A second opportunity arises when the collective cannot be removed at all: a baseline may prepare each shard in a Python loop before the collective even though the same work is one vectorized operation, so the rewrite leaves the collective count unchanged but eliminates repeated per-shard processing. These cases illustrate a broader gap: some optimizations require reasoning about the computation surrounding a collective, not only selecting, configuring, or rearranging primitives.

We address this gap with SorcarCCL, a research-agent pipeline inspired by Sorcar, the research agent of the KISS framework (Sen, 2026). At its center is a budget-tracked ReAct loop (Yao et al., 2023) whose system prompt specifies a research procedure—profile the baseline, gather prior approaches, propose and pairwise-judge ideas, implement and evaluate the most promising candidate, and stop once the target is met (Section 3.2)—together with an adversarial-testing pass in which one subtask constructs inputs intended to break a candidate and another repairs it. The agent receives a 37-line task prompt, a `score_candidate` tool that returns the simulator's estimate, and a `read_reference` tool that returns a 197-line Trainium reference (the same reference OverlayCCL included in every prompt). We keep the same target hardware, simulator, and proposer model as OverlayCCL, but extend the simulator with a standalone-graph cost branch and probes for unsupported operators (Section 3.1); the simulator is calibrated deterministically before search and requires no LLM tokens.

Our contributions are the following.

1. A research-agent framework that extends collective optimization beyond structural rewrites, searching more complex optimization opportunities under the same simulator and hardware constraints as OverlayCCL.
2. A production-derived benchmark of eight families of collective-communication optimizations from AWS data-parallel and mixture-of-experts workloads, exposing opportunities not captured by structural search, including cross-operation algebraic identities and vectorization of per-shard processing.
3. An empirical study of the resulting search space: across the OverlayCCL problems SorcarCCL introduces no regression, and across 40 production problems it consistently discovers additional improvements, with statistical validation across nine search seeds and hardware validation on 224 Trainium cores.
4. A case study of the improvements' source: a mixture-of-experts routing reduction in which an algebraic identity reduces the number of collectives ($4.06\times$), together with a complementary per-block normalization in which vectorization removes repeated processing without changing the collective count ($4.31\times$).

**Scope.** We use the same closed stack, calibrated simulator, hardware gates, and proposer model (Claude Sonnet 4.5, Anthropic, 2025) as OverlayCCL (Ke et al., 2026), so that the comparison isolates the search procedure. The baseline is the OverlayCCL-optimized AWS production recipe on these patterns. Each problem is searched under nine seeds; the confirmed median-seed rewrite is then evaluated on hardware with a single warm-cache measurement. We also integrate the emitted rewrites into end-to-end GPT- and Llama-style transformer training (Section 4.4).

## 2 Problem Setting

**Hardware and vendor boundary.** All measurements run on 7 AWS `trn1.32xlarge` instances in a single placement group, with 32 NeuronCores per node and 224 ranks in total. Communication uses NeuronLink, the on-instance accelerator interconnect, within each node and eight Elastic Fabric

Adapters (EFA, AWS's RDMA-capable cluster network interface) per node across nodes. The Neuron runtime compiles each PyTorch/XLA graph into a NEFF (Neuron Executable File Format) binary and executes collectives using closed-source implementations. A candidate can invoke the five `xm.*` collective primitives and control the exposed configuration parameters, but cannot control how a collective moves bytes internally.

**Collective problems.** Each problem consists of a Python function signature, a formula defining the output tensor from the input, rank, and world size, a reference implementation, and a set of admissible shapes. The eight problems from Ke et al. (2026) cover the training collectives in their evaluation, from mixture-of-experts all-to-all to bucketed gradient all-reduce; SorcarCCL matches OverlayCCL on all eight with no regression (Section 4.1). The 40 problems studied in detail are drawn from AWS production data-parallel and mixture-of-experts patterns and use all-reduce, all-gather, and reduce-scatter across the eight families in Table 1. Each admits multiple plausible implementations, and the specification does not identify an optimal rewrite.

**Three code paths.** For every problem we generate three implementations: the *baseline* AWS production recipe, the *OverlayCCL* output of OverlayCCL's strategy-enumeration search, and the *SorcarCCL* output of the SorcarCCL research loop (Section 3); both searches use the same underlying simulator and hardware constraints. Either may transform the source through loop accumulation, per-tensor processing, payload re-bucketing, or changes to the composition or choice of collective primitives. Every candidate must compile and pass a lightweight training-validation gate that checks output equivalence before performance evaluation.

**Simulator and hardware evaluation.** The simulator returns an estimated step time, `sim_time_us`, from a cost model fitted to probes on the target cluster (Section 3.1) and ranks candidates during search. Every confirmed improvement is validated with a warm-cache hardware measurement: each variant runs twice consecutively and we report the second run (20 warm-up then 100 timed iterations), avoiding the cold NEFF compilation cost of the first run, which inflates step time by roughly an order of magnitude. A simulator improvement not confirmed on hardware is treated as a tie, so each reported hardware result is a simulator-selected candidate with its hardware measurement.

## 3 The System

Figure 1 shows the three phases. **Phase 1** builds the simulator: a deterministic probe campaign on the target cluster fits the constants of the cost model. **Phase 2** performs the search: starting from a seed library of reference implementations, the research agent proposes and scores candidates using the Phase-1 simulator, replacing OverlayCCL's strategy-enumeration search. **Phase 3** gates and emits the result: every surviving candidate must compile and run at the training world size, after which the lowest-cost survivor is emitted as a runtime module.

Phase 1 follows OverlayCCL's calibration but replaces its LLM-generated probes with deterministic ones that additionally cover a standalone-graph cost branch and unsupported operators (Section 3.1), and Phase 3 keeps its compile-and-run validation at training scale. Phase 2 is the main methodological departure: rather than enumerating predefined strategy transformations and refining each iteratively, the SorcarCCL research agent proposes, selects, implements, evaluates, and refines optimization ideas with a ledger recording lessons learned.

### 3.1 Phase 1: deterministic calibration

Phase 1 runs a fixed probe campaign on the target cluster and fits the constants of a six-term cost model: per-operation local cost, per-collective bandwidth, amortization of consecutive dispatches, graph-launch overhead, NEFF reload cost, and network transfer. The probes sweep synthetic tensors over size ladders and measure only the machine and collective library, so one set of constants scores every problem. OverlayCCL uses an LLM to select which probes to issue, costing tens of minutes of agent turns per invocation; we instead run the complete campaign deterministically, in seconds and with no token cost. Despite using only synthetic probes, the resulting simulator selects the same per-problem ranking as hardware measurements on our evaluation set, and, as in OverlayCCL,

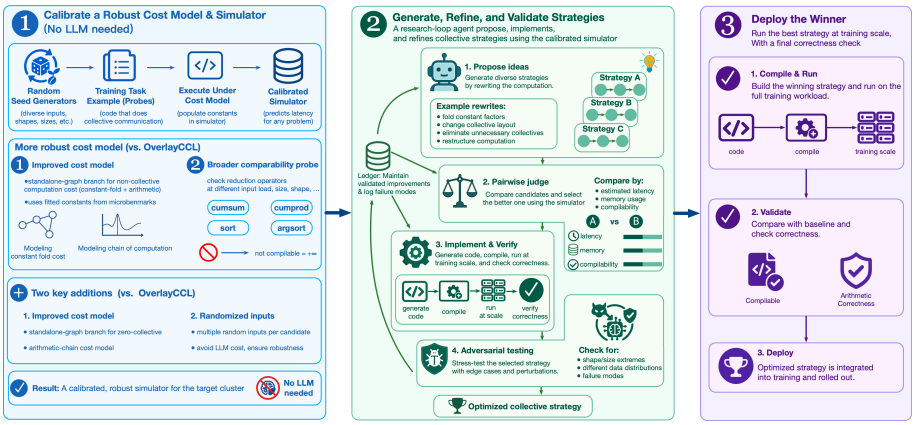


Figure 1: The three-phase SorcarCCL pipeline. Phase 1 calibrates the cost-model simulator with a deterministic probe campaign (no LLM); Phase 2 runs the research loop—profile, propose and pairwise-judge, adversarially test, score, log—which OverlayCCL replaces with a fixed strategy enumeration; Phase 3 gates every survivor at 224 ranks and emits the lowest-cost one.

its predicted speedup ratios are generally conservative, underestimating the hardware speedups (Appendix A).

**Why a fixed probe campaign is faithful.** Each probe measures a property of the machine and collective library rather than the training task: collective cost depends on payload size, rank count, and dispatch pipelining, not on tensor values or the model that produced them. Sweeping these factors across diverse inputs captures the cost dimensions needed to compare implementations across workloads, from mixture-of-experts all-to-all to dense gradient all-reduce. Hardware and software drift is handled at calibration time—the campaign is rerun whenever the instance type or SDK changes, re-deriving all constants from fresh measurements.

We add two extensions to the cost model. First, the original simulator was calibrated primarily on collective communication and does not accurately capture the preprocessing and postprocessing around it; we add a separate model for these local computations, fitted from microbenchmarks in the same probe campaign, so the simulator does not systematically underestimate candidates that eliminate communication but add local computation. Second, a compilability probe tests operators that some Neuron compiler versions do not support (e.g. `cumsum`, `cumprod`, `sort`, `argsort` for v2.26); candidates that use them receive a score of $+\infty$.

### 3.2 Phase 2: The Research-Loop Agent

**The agent.** Phase 2 uses a ReAct loop with native function calling, capped at 30 steps and 5 USD of Claude Sonnet 4.5 usage per problem (same constraints as OverlayCCL). The system specifies two complementary disciplines. In the *research loop*, the agent profiles the baseline and records its metric, gathers prior approaches, proposes candidate ideas with rationales, judges them pairwise, implements and evaluates the most promising one, and keeps or rejects it by the result; it then generates new ideas while excluding logged failures and stops when the target is reached. The *adversarial-testing* discipline asks one subtask to construct inputs intended to break a candidate and another to repair it. Algorithm 1 in Appendix C gives the complete loop; the tools it invokes are described below.

Each proposed rewrite must preserve the specified function while reducing modeled cost: for a candidate $g$ relative to $f$, the target is $g \equiv f$ under the correctness oracle and $\mathrm{Sim}(g) < \mathrm{Sim}(f)$. The agent may exploit all-reduce linearity ($\sum_i c_i \,\mathrm{AR}(x_i) = \mathrm{AR}(\sum_i c_i x_i)$, replacing $k$ all-reduces with one), or leave the collective count unchanged and instead alter the surrounding processing, for example replacing a per-block-statistics loop with a single `view` + reduction over a batched axis. The correctness oracle $\mathcal{O}$ and the hardware gate reject rewrites that appear cheaper but do not preserve the required output.

**The tools.** The agent receives `score_candidate`, `read_reference`, and a `finish` action. `score_candidate(code)` evaluates a candidate at small world sizes using rank-asymmetric inputs and returns `sim_time_us`. `read_reference()` returns a 197-line Trainium reference describing the XLA collective API, unsupported primitives, simulator cost terms, and several Trainium specific quirks. This is the same Trainium-specific knowledge that OverlayCCL places directly in its prompt. Moving it behind a tool keeps the standing prompt short and lets the agent retrieve it only when needed. The document contains no per-problem hint or rewrite-family label.

**The correctness oracle.** The prompt asks the agent to produce the 'exact tensor" specified by the formula. Because different implementations can change floating-point reduction order, we operationalize 'exact" as agreement with a gold reference within a fixed tolerance rather than bitwise equality. The oracle runs the candidate for every rank at world sizes 4 and 8 in one process and compares each rank's output against the reference on a fixed sample of admissible shapes; inputs are deterministic and rank-distinguishable, so rewrites that drop or double-count communication fail. The comparison uses `allclose` with $\text{atol} = \text{rtol} = 10^{-3}$ in fp32, plus a bf16 check at world size 8 ($\text{atol} = 0.1$, $\text{rtol} = 0.05$) for reduction-order effects; shape mismatches are rejected first and any NaN or Inf is a failure. Phase 3 (Section 3.3) then validates the survivor at the full 224 ranks. This oracle, not the wording of the prompt, defines correctness.

**What the ledger records.** The ledger records the baseline metric (the original implementation's `sim_time_us`) and a history of the ideas explored in Phase 2—both those rejected during pairwise judging and those implemented and evaluated, with their `score_candidate` results (e.g. folding $k$ all-reduces, hoisting a repeated call, or replacing an all-reduce-then-slice with a reduce-scatter). This history is fed back during later rounds of idea generation so the agent avoids re-exploring ideas already rejected or shown ineffective.

**Adversarial testing.** Before scoring a rewrite, the prompt asks the agent to design small-scale tests intended to expose differences between the candidate and the baseline. We provide no test cases or hints; the agent must itself find inputs that could make an incorrect rewrite diverge from the reference. In practice, trajectories develop tests involving rank-dependent factors, non-uniform shapes, and reduction-operator corner cases such as idempotent `MAX` versus linear `SUM`. This is an inexpensive correctness filter ahead of the costlier Phase 3 gate, so candidates that are fast in the simulator but fail these tests are rejected before any compilation or training run.

### 3.3 Phase 3: gates and emission

Every Phase-2 survivor must compile and run at the target world size on the cluster (gate 3a) and inside an 8-layer language-model step (gate 3b); a crash, out-of-memory error, NaN, or shape failure removes the candidate regardless of its simulator score. Among the survivors, the one with the lowest `sim_time_us` is emitted as a drop-in runtime module. Hardware time serves as a validity gate rather than the search objective, because isolated microbenchmarks can rank candidates differently from training-scope measurements on this stack (Ke et al., 2026).

## 4 Evaluation

**Setup.** All searches use Claude Sonnet 4.5 through Amazon Bedrock (model `claude-sonnet-4-5-20250929`) as the proposer, the calibrated simulator from Section 3.1, and a per-problem budget of 30 steps and 5 USD. OverlayCCL uses its original fixed search protocol. Because OverlayCCL's authors do not report results on these problems, the OverlayCCL numbers are obtained by rerunning its search through our shared evaluation pipeline. Experiments use Neuron SDK 2.x images with PyTorch/XLA 2.8 or 2.9 and Neuron compiler 2.23 or 2.26; the released scripts record the software versions for each run.

### 4.1 No regression on OverlayCCL's own problems

We first test whether replacing OverlayCCL's strategy-enumeration proposer introduces any regression on the problems OverlayCCL was designed for. On its eight evaluation problems—mixture-of-experts

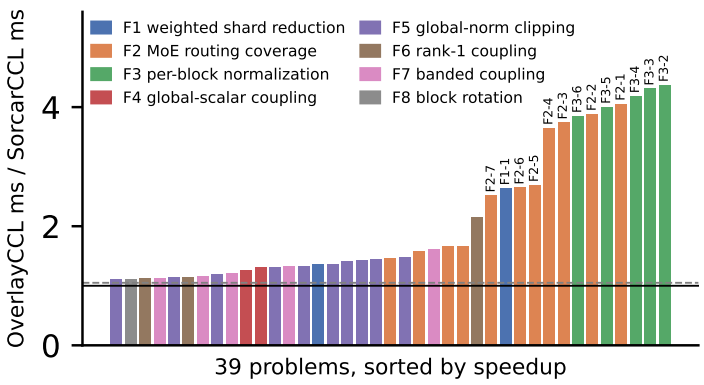


Figure 2: Warm-cache speedup of the median-seed SorcarCCL rewrite over the OverlayCCL output at 224 ranks, sorted and colored by family; the dashed line is the $1.05\times$ threshold and bars above $2.5\times$ carry the problem name. One F3 problem exposes an OverlayCCL compile-time pathology ($117\times$) and is clipped off the axis.

all-to-all and unequal all-to-all, ring key-value exchange, bucketed gradient all-reduce, pipeline send-receive, tensor-parallel MLP, sharded-parameter prefetch, and layer-block all-reduce—SorcarCCL recovers the same schedule within warm-cache hardware noise on all eight. None of the eight rewrite families applies here: these optimization opportunities are choices of primitive and data layout that OverlayCCL's enumeration already explores. The released scripts reproduce the three-way runtime comparison for all eight.

### 4.2 The 40 problems on hardware

We obtained 40 collective-communication routines from AWS production recipes; the full problem set, both controllers' emitted code for every seed, and scripts that reproduce every table and figure are publicly available for reproduction (SorcarCCL, 2026). Across the 40 problems, the simulator gap between OverlayCCL and SorcarCCL ranges from $1.00\times$ to $1.28\times$ over the 35 problems with a recorded median-seed ratio; for small collectives both implementations approach the simulator's dispatch-time floor, so the modeled gap can be modest even when the hardware gap is larger. Each problem is searched with nine independent seeds, and we count a win only when the best-of-nine simulator ratio is at least $1.05\times$, the one-sided Mann–Whitney U test gives $p < 0.05$, and the bootstrap 95% lower bound exceeds 1.0; all 40 problems qualify. Warm-cache measurements of the median-seed rewrite then confirm a speedup of at least $1.05\times$ on all 40 problems (Table 1, Figure 2). Excluding one F3 problem with an OverlayCCL compile-time pathology (a $117\times$ outlier), the confirmed speedups range from $1.12\times$ to $4.37\times$, with median $1.46\times$ and geometric mean $1.86\times$; across all 40 paired measurements, OverlayCCL is faster than SorcarCCL on 0 cases (Appendix A).

To understand where the two searches differ, we manually inspect the median-performing output from all nine seeds for each of the 40 problems and categorize the differences into eight rewrite families (Table 1), all occurring among the 40 problems. Six families (F1, F2, F4, F5, F6, F7) reduce the number of collective operations—not by repackaging or reordering existing data, but by recognizing that several collectives compute the same result as a smaller set (e.g. multiple all-reduces collapse to one after rewriting the local computation). The other two (F3, F8) keep the collective count unchanged but replace repeated per-block processing with a vectorized operation.

These families expose a limitation of OverlayCCL's enumerate-then-refine workflow. Transformations such as concatenating tensors before an all-reduce or moving a reshape outside a collective fit predefined templates because they preserve the communication pattern; the eight families instead require reasoning about the computation the collective sequence performs, establishing that a seemingly different program is mathematically equivalent before removing a collective or rewriting the surrounding work. Three properties make this hard for fixed templates. The equivalence can span a whole sequence rather than one operation: a loop may issue one collective per iteration while

Table 1: The eight rewrite families where SorcarCCL beats OverlayCCL, with problem count and median/best warm-cache hardware speedup at 224 ranks over the OverlayCCL output (which tracks the AWS production baseline). One F3 problem triggers an OverlayCCL compile-time pathology ($117\times$) and is excluded; per-problem numbers in Appendix A.

| | Family | OverlayCCL strategy | SorcarCCL strategy | # | RT med | RT best |
|---|---|---|---|---|---|---|
| F1 | Weighted per-shard reduction | $L$ weighted all-reduces $\sum_i c_i \,\mathrm{AR}(x_i)$ | $\mathrm{AR}(\sum_i c_i x_i)$: one all-reduce, local post-scale | 2 | 2.00× | 2.64× |
| F2 | MoE routing coverage | $L$ masked all-reduces, renormalized per expert | coverage counts are constants; one masked all-reduce | 11 | 2.65× | 4.06× |
| F3 | Per-block normalization | per-block RMS/variance in a Python slice loop | irreducible count; `view(B,S)+mean(dim=1)` | 6 | 4.19× | 4.37× |
| F4 | Global-scalar coupling | scalar and payload reductions as separate collectives | one combined collective | 2 | 1.28× | 1.30× |
| F5 | Global-norm clipping | squared-norm and payload reductions per sub-step | compose into fewer collectives | 10 | 1.35× | 1.47× |
| F6 | Rank-1 coupling | all-reduce of the full $A + vv^\top$ matrix | reduce the vector, reconstruct locally | 3 | 1.15× | 2.15× |
| F7 | Banded neighbor coupling | one all-reduce per diagonal/off-diagonal round | band composes; one reduction over the payload | 5 | 1.22× | 1.61× |
| F8 | Block rotation | per-block $2 \times 2$ rotation in a loop | rotations compose; vectorized application | 1 | 1.12× | 1.12× |

the equivalent computation needs only a few after algebraic simplification. Intermediate values can depend on rank or routing pattern, so the two computations appear different on each device even when the differences cancel. And the simplification is not always valid for every instance or loop depth, so the agent must derive and verify the transformation rather than apply a fixed rule. Hence the difference between the procedures: OverlayCCL rearranges the existing communication structure, whereas SorcarCCL proposes a new computation, tests its equivalence, and retains it only when the correctness and performance checks succeed.

The largest gains came from the per-block-normalization and routing families (F3-2 at 4.37×; F2-1 at 4.06×, the case study of Section 4.3) and the smallest from families that remove only a handful of dispatches (F4 at 1.30×, F8 at 1.12×), since one dispatch costs about 0.4 ms at 224 ranks. Appendix D shows, for one problem in each family, the OverlayCCL and SorcarCCL programs side by side and explains why the strategy enumeration stops at a local optimum; we name problems by a per-family ID (F3-2 is the second F3 problem), which Appendix A maps to its log name.

### 4.3 Case Study: Mixture-of-Experts Routing Coverage

The first problem in Family 2 (F2-1) aggregates a mixture-of-experts routing statistic over $L{=}5$ rounds on a $d{=}8$ shard grid with $b{=}16$ expert blocks. Each rank routes to a contiguous window of experts, so every block $b$ is covered by some number of ranks $c_b$—its *coverage count*. In each round the baseline masks each rank's contribution to the blocks it covers, all-reduces the masked payload, and renormalizes every block by $c_b$ before the next round; a final all-reduce closes the aggregation. The baseline therefore issues eight all-reduces, one per masked round plus the closing reduction (Appendix D.2).

The apparent obstacle is that the per-block renormalization by $c_b$ sits between successive collectives, so each round looks like a data-dependent transformation that must be materialized before the next begins. But the coverage counts $c_b$ are fixed by the routing table, not by tensor values, so they are step-invariant constants, and masking followed by all-reduce is linear. The mask–reduce–renormalize recurrence therefore composes: the constant renormalizations telescope out and the whole cascade collapses to a single masked all-reduce on the once-reduced payload. SorcarCCL recognizes this and replaces the eight collectives with two, for a $4.06\times$ hardware speedup over OverlayCCL's output.

OverlayCCL's enumeration can fuse structurally redundant collectives but never removes the intervening coverage-count renormalization, so its refinements stay near the local optimum of one all-reduce per round. The research loop instead reasons about the computation between collectives, finding that the constant coverage counts commute through the reductions—an improvement from discovering a new equivalent computation with fewer collectives, not from selecting a better predefined strategy.

The loop also finds a complementary rewrite that leaves the collective count unchanged. In the per-block normalization of Family 3 (F3-3, same $d{=}8$ grid) the collectives are irreducible—each

round reduces the payload, scales block $b$ by a data-dependent factor $f_b = 1 + 3\,\mathbb{E}[s_b^2]$, and reduces again. OverlayCCL marshals the per-block factors in an unrolled Python loop that lowers to scalar-indexed slice operations, whereas SorcarCCL reshapes the payload to $(b, S)$ and computes all factors at once with a single `mean(dim=1)` and broadcast multiply: a $4.31\times$ speedup with no change in collective count, because the scalar loop does not vectorize on the device while the reshaped form does (Appendix D.3). The two mechanisms together—collective-count reduction and local-computation vectorization—show the loop's reach beyond OverlayCCL's strategy enumeration.

### 4.4 End-to-end training

The results above measure each rewrite in isolation. To evaluate their impact in a full training run, we follow prior work on collective-communication libraries (Cai et al., 2021; Cowan et al., 2023; Shah et al., 2023; Ke et al., 2026) and integrate the emitted rewrites into end-to-end training of 9.7-billion-parameter GPT-3- and Llama-style transformers ($48$ layers, model dimension $4096$, $64$ attention heads) on all 224 ranks with TP$=32\times$DP$=7$, in the regime OverlayCCL identified on OLMoE where the active parameter count is small enough that collective communication dominates step time (Ke et al., 2026; Muennighoff et al., 2025). Each emitted family rewrite replaces the corresponding tensor-parallel collective implementation. Over 120 steps, the SorcarCCL rewrites reduce per-step time by $2.13\times$ for the Llama-style model and $2.15\times$ for the GPT-3-style model.

The model loss stays bit-identical to the OverlayCCL backend at every step, so the comparison isolates step-time effects; functional equivalence is validated independently by the fp32 correctness oracle (Appendix B).

### 4.5 Cost analysis

On average, OverlayCCL cost more per problem: 512 model calls against 168 ($3.05\times$), 354,590 output tokens against 106,007, and fewer input tokens (818,558 against 1,259,442), for an estimated 7.77 USD against 5.37 USD at Sonnet 4.5 list prices, or 45% more.

### 4.6 Methodology ablation

Disabling one stage of the SorcarCCL research loop at a time and re-running the search on the 40-problem set (fixing simulator, proposer model, and budgets), every ablated arm is net-negative relative to the complete loop—idea proposal and iterative refinement are the most load-bearing—and the emitted code shows the regressions are structural: the full loop collapses an iterated-collective loop to two all-reduces where an ablated arm keeps one collective per iteration, and no ablation finds a rewrite the full loop cannot (Appendix C). An ablation of the simulator extensions (Section 3.1) is equally sharp: without the standalone-graph cost branch the simulator scores a candidate that keeps a provably dead all-reduce almost identically to one that removes it, so the agent ships the dead collective in half its runs—a $36\times$ hardware regression.

## 5 Related Work

We organize prior work by the layer it optimizes and how it searches.

**Synthesis below the library boundary.** SCCL synthesizes Pareto-optimal collective algorithms for a target topology (Cai et al., 2021); MSCCLang compiles a chunk-level language into NCCL-executed schedules (Cowan et al., 2023); TACCL guides synthesis with communication sketches (Shah et al., 2023); TACOS extends synthesis to arbitrary topologies (Won et al., 2024); and AutoCCL tunes NCCL's algorithm, protocol, channel, and chunk parameters (Xu et al., 2025). CoCoNet fuses computation into communication kernels (Jangda et al., 2022), and Centauri partitions collectives to overlap communication with computation via fixed primitive substitutions (Chen et al., 2024). All optimize how an individual collective moves data and need a runtime that exposes the corresponding schedule or configuration; Neuron exposes no such control, so we hold each collective's implementation fixed and optimize the sequence of calls around it.

**Simulators.** ASTRA-sim 2.0, its 3.0 revision, and SimAI predict collective and training performance from network and system models calibrated to reference configurations (Won et al., 2023; 2026; Wang et al., 2025). Our simulator uses a coarser six-term cost model but is re-fit directly on the deployment cluster through deterministic probes and checked against hardware.

**Hand-written rewrites.** The rewrite families in Table 1 have precedents in manually designed training systems: PyTorch DDP and Horovod bucket gradients into fewer all-reduces (Li et al., 2020; Sergeev & Del Balso, 2018); ZeRO and FSDP replace gradient all-reduce with reduce-scatter and all-gather (Rajbhandari et al., 2020; Zhao et al., 2023); and Megatron-LM uses tensor-parallel all-reduces in the paths studied here (Shoeybi et al., 2019)—all encoded explicitly rather than discovered by search. Classical compiler passes such as common-subexpression and dead-code elimination (Aho et al., 2006) and XLA operator fusion (Snider & Liang, 2023) remove redundant computation but do not address the cross-collective algebraic rewrites in F1, F2, and F4, where equivalence depends on the relationship among multiple collective calls. The all-to-all patterns of GShard (Lepikhin et al., 2021), expert-choice routing (Zhou et al., 2022), and OLMoE (Muennighoff et al., 2025) motivate our MoE workloads, while our families target the aggregation and normalization around these exchanges.

**Program search with language models.** FunSearch combines a language model with an exact evaluator in an evolutionary search loop (Romera-Paredes et al., 2024); AlphaEvolve extends it to larger codebases (Novikov et al., 2025); and ShinkaEvolve reduces the samples needed for evolutionary code search (Lange et al., 2025). Kernel-oriented systems search CUDA programs against benchmarks such as KernelBench (Ouyang et al., 2025), including multi-turn (Baronio et al., 2025) and contrastive (Li et al., 2026) reinforcement learning. These optimize program text against an executable scorer, but such an objective can itself fail: Sakana AI's kernel agent discovered a harness exploit that bypassed correctness checking (Sakana AI, 2025), illustrating reward hacking (Skalse et al., 2022). SorcarCCL separates correctness from performance with an explicit oracle and an adversarial pass that rejects incorrect candidates before scoring, generating rank-asymmetric inputs analogous to property-based testing such as QuickCheck (Claessen & Hughes, 2000).

**Agent loops.** ReAct interleaves reasoning with tool calls (Yao et al., 2023), while Reflexion and Self-Refine guide later attempts with feedback from earlier ones (Shinn et al., 2023; Madaan et al., 2023). SorcarCCL instantiates this pattern for collective-communication optimization with an executable scorer as the primary feedback (Sen, 2026); its prescribed loop—profile, propose, judge, implement, adversarially test—lets the agent formulate and validate rewrites outside OverlayCCL's strategy-enumeration space.

## 6 Limitations

The 40 problems are centered on AWS production data-parallel and mixture-of-experts patterns, so these results do not estimate how often the two pipelines would differ on arbitrary collective-communication problems. The nine-seed confirmation is performed in the simulator; because Trainium hardware is costly, we take only the median-performing seed of each confirmed rewrite to hardware and do not report hardware variance across seeds, though every simulator-confirmed improvement remains an improvement on hardware. The end-to-end runs (Section 4.4) likewise use one training run per configuration.

## 7 Conclusion

Optimizing collective communication above a closed library requires reasoning beyond the choice of primitive and payload layout: OverlayCCL searches these structural choices, but under the same simulator and correctness gates only SorcarCCL also finds the algebraic rewrites that remove redundant collectives and the vectorization around an unavoidable collective—improvements robust across nine seeds and confirmed on hardware for all 40 production problems (median 1.46×, OverlayCCL faster on 0), at no calibration token cost.

### REPRODUCIBILITY STATEMENT

Section 2 gives the hardware, SDK, world size, and warm-cache measurement rule; Section 3.1 gives the simulator; Appendix A lists every problem with its simulator and hardware times. Every number in the paper is a macro; the comment beside each names the result table it comes from or the key of the script that recomputes it from the result tables, and the results figure and the appendix tables are generated by scripts from the same tables. The source code, the emitted candidates for every seed, and the result logs are available in an anonymized artifact repository (SorcarCCL, 2026), together with a README describing the Trainium cluster environment and the commands that reproduce each table and figure.

### AI USE STATEMENT

Large language models are the object of study: the collective rewrites evaluated here were produced by our SorcarCCL-inspired pipeline and by OverlayCCL, both driven by Claude Sonnet 4.5, as Sections 3 and 4 describe. We also used generative AI agents to interpret the result logs and propose claims from them, to generate the table scripts, and to check the text against the result tables. We did not use generative AI to generate or alter any measurement. We ran all AI-written code on the hardware described and report its outputs; we checked every reference against its publisher or arXiv page; we read and edited every sentence and decided which claims to keep. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

## REFERENCES

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2nd edition, 2006.

Amazon Web Services. What is Neuron collective communication? AWS Neuron Documentation, Neuron SDK 2.32, 2026. URL `https://awsdocs-neuron.readthedocs-hosted.com/en/latest/neuron-runtime/about/collectives.html`. Accessed 2026.

Anthropic. Introducing Claude Sonnet 4.5. `https://www.anthropic.com/news/claude-sonnet-4-5`, September 2025. Published September 29, 2025. Accessed September 2026.

Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. Kevin: Multi-turn RL for generating CUDA kernels, 2025.

Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. Synthesizing optimal collective algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '21)*, pp. 62–75, New York, NY, USA, 2021. ACM. doi: 10.1145/3437801.3441620.

Chang Chen, Xiuhong Li, Qianchao Zhu, Jiangfei Duan, Peng Sun, Xingcheng Zhang, and Chao Yang. Centauri: Enabling efficient scheduling for communication-computation overlap in large model training via communication partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24)*, pp. 178–191, New York, NY, USA, 2024. ACM. doi: 10.1145/3620666.3651379.

Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pp. 268–279. ACM, 2000. doi: 10.1145/351240.351266.

Meghan Cowan, Saeed Maleki, Madanlal Musuvathi, Olli Saarikivi, and Yifan 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 '23)*, pp. 502–514, New York, NY, USA, 2023. ACM. doi: 10.1145/3575693.3575724.

Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. Breaking the computation and

communication abstraction barrier in distributed machine learning workloads. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*, pp. 402–416, New York, NY, USA, 2022. ACM. doi: 10.1145/3503222.3507778.

Kaiyao Ke, Ziming Mao, Ion Stoica, and Koushik Sen. OverlayCCL: Composing collectives above a vendor library via a closed-stack search loop. Manuscript, https://github.com/SorcarCCL/sorcarccl-artifact/blob/main/docs/overlayccl.pdf, 2026. Accessed 2026.

Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. ShinkaEvolve: Towards open-ended and sample-efficient program evolution, 2025.

Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. GShard: Scaling giant models with conditional computation and automatic sharding. In *International Conference on Learning Representations (ICLR)*, 2021. URL https://openreview.net/forum?id=qrwe7XHTmYb.

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, 2020. doi: 10.14778/3415478.3415530.

Xiaoya Li, Xiaofei Sun, Albert Wang, Jiwei Li, and Chris Shum. CUDA-L1: Improving CUDA optimization via contrastive reinforcement learning. In *The Fourteenth International Conference on Learning Representations (ICLR)*, 2026. URL https://openreview.net/forum?id=igZItUbY6n.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems 36 (NeurIPS)*, 2023. doi: 10.52202/075280-2019.

Niklas Muennighoff, Luca Soldaini, Dirk Groeneveld, Kyle Lo, Jacob Morrison, Sewon Min, Weijia Shi, Pete Walsh, Oyvind Tafjord, Nathan Lambert, Yuling Gu, Shane Arora, Akshita Bhagia, Dustin Schwenk, David Wadden, Alexander Wettig, Binyuan Hui, Tim Dettmers, Douwe Kiela, Ali Farhadi, Noah A. Smith, Pang Wei Koh, Amanpreet Singh, and Hannaneh Hajishirzi. OLMoE: Open mixture-of-experts language models. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://openreview.net/forum?id=xXTkbTBmqq.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery, 2025.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. KernelBench: Can LLMs write efficient GPU kernels?, 2025.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, Atlanta, GA, USA, 2020. IEEE. doi: 10.1109/SC41405.2020.00024.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625:468–475, 2024. doi: 10.1038/s41586-023-06924-6.

Sakana AI. The AI CUDA engineer: Agentic CUDA kernel discovery, optimization and composition. Blog post, https://sakana.ai/ai-cuda-engineer/,

archived at https://web.archive.org/web/20250223061804/https://sakana.ai/ai-cuda-engineer/, February 2025. Updated February 21, 2025 with a reward-hacking addendum. Accessed September 2026.

Koushik Sen. KISS Sorcar: A stupidly-simple general-purpose and software engineering AI assistant, 2026.

Alexander Sergeev and Mike Del Balso. Horovod: Fast and easy distributed deep learning in TensorFlow, 2018.

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 23)*, pp. 593–612, Boston, MA, USA, 2023. USENIX Association.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems 36 (NeurIPS)*, 2023. doi: 10.52202/075280-0377.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism, 2019.

Joar Skalse, Nikolaus H. R. Howe, Dmitrii Krasheninnikov, and David Krueger. Defining and characterizing reward gaming. In *Advances in Neural Information Processing Systems 35 (NeurIPS)*, 2022. doi: 10.52202/068431-0687. arXiv version titled "Defining and Characterizing Reward Hacking", arXiv:2209.13085.

Daniel Snider and Ruofan Liang. Operator fusion in XLA: Analysis and evaluation, 2023.

SorcarCCL. SorcarCCL: Production collective-communication problem set, code-generation results, and reproduction scripts. https://github.com/SorcarCCL/sorcarccl-artifact, 2026. Anonymized artifact repository.

Xizheng Wang, Qingxu Li, Yichi Xu, Gang Lu, Dan Li, Li Chen, Heyang Zhou, Linkang Zheng, Sen Zhang, Yikai Zhu, Yang Liu, Pengcheng Zhang, Kun Qian, Kunling He, Jiaqi Gao, Ennan Zhai, Dennis Cai, and Binzhang Fu. SimAI: Unifying architecture design and performance tuning for large-scale large language model training with scalability and precision. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pp. 541–558, Philadelphia, PA, USA, 2025. USENIX Association.

William Won, Taekyung Heo, Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. ASTRA-sim2.0: Modeling hierarchical networks and disaggregated systems for large-model training at scale. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 283–294, Raleigh, NC, USA, 2023. IEEE. doi: 10.1109/ISPASS57527.2023.00035.

William Won, Midhilesh Elavazhagan, Sudarshan Srinivasan, Swati Gupta, and Tushar Krishna. TACOS: Topology-aware collective algorithm synthesizer for distributed machine learning. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 856–870, Austin, TX, USA, 2024. IEEE. doi: 10.1109/MICRO61859.2024.00068.

William Won, Jinsun Yoo, Tuan Ta, Moumita Dey, Andy Balogh, Pradosh Datta, Furkan Eris, Conor Green, Winston Liu, Changhai Man, Kingshuk Mandal, Amos Rai, Vinay Ramakrishnaiah, Ruchi Shah, David Sidler, Harsh Sikhwal, Hanjiang Wu, Tushar Krishna, and Bradford M. Beckmann. ASTRA-sim 3.0: Next-level distributed machine learning simulations via high-fidelity GPU and infrastructure modeling, 2026.

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 25)*, pp. 667–683, Philadelphia, PA, USA, 2025. USENIX Association.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations (ICLR)*, 2023.

Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. PyTorch FSDP: Experiences on scaling fully sharded data parallel. *Proceedings of the VLDB Endowment*, 16(12):3848–3860, 2023. doi: 10.14778/3611540.3611569.

Yanqi Zhou, Tao Lei, Hanxiao Liu, Nan Du, Yanping Huang, Vincent Zhao, Andrew M. Dai, Zhifeng Chen, Quoc V. Le, and James Laudon. Mixture-of-experts with expert choice routing. In *Advances in Neural Information Processing Systems 35 (NeurIPS 2022)*, 2022. doi: 10.52202/068431-0515.

## A THE PROBLEM POOL

Tables 2–5 list every problem of the 40-problem pool: Tables 2 and 3 give the median-seed simulator times of the OverlayCCL and SorcarCCL candidates and Tables 4 and 5 the single-seed warm-cache hardware times of both, each split by family (F1–F4 and F5–F8) at 224 ranks. Each problem is named by a sequential per-family ID (`F3-2` is the second F3 problem, numbered in descending order of hardware speedup); the second column of each table gives the log name that ID maps to in the released logs, so the tables are themselves the ID-to-log-name map used throughout the paper. A dashed simulator cell marks a hardware-measured row with no recorded median-seed ratio. This set is the 40 problems only; problems on which OverlayCCL and SorcarCCL agree are not reported here. All tables are generated from the result tables by `make_tables.py`.

Table 2: Production problem set, families F1–F4 — median-seed simulator time ($\mu$s) of each controller's candidate at 224 ranks. A dashed cell marks a hardware-measured row with no recorded median-seed ratio.

| ID | log name | sim OverlayCCL ($\mu$s) | sim SorcarCCL ($\mu$s) | sim ratio |
|---|---|---|---|---|
| *F1 weighted per-shard gradient reduction* | | | | |
| F1-1 | `r71_wdiag_b6_L5_w1_d8` | 6709 | 6627 | 1.01× |
| F1-2 | `r71_wdiag_b5_L3_w1_d8` | 6670 | 6619 | 1.01× |
| *F2 MoE routing / expert-assignment coverage* | | | | |
| F2-1 | `r60d_b16_L5_p2048_d8` | 6907 | 5378 | 1.28× |
| F2-2 | `r60d_b10_L5_p2048_d8` | 6729 | 5374 | 1.25× |
| F2-3 | `r60c_b12_L5_p2048_d8` | 6812 | 5376 | 1.27× |
| F2-4 | `r60d_b13_L4_p2048_d8` | 6828 | 5403 | 1.26× |
| F2-5 | `r43_route_B9_d8_L3_count8` | – | – | – |
| F2-6 | `r40_route_d8_L3_count8` | – | – | – |
| F2-7 | `r60c_b8_L2_p2048_d8` | 6696 | 5366 | 1.25× |
| F2-8 | `r60c_b10_L4_p2048_d8` | 6757 | 6632 | 1.02× |
| F2-9 | `r60_b5_L2_d8` | 6625 | 6598 | 1.00× |
| F2-10 | `r40_route_d6_L3_count8` | – | – | – |
| F2-11 | `r60c_b9_L3_p2048_d8` | 6726 | 5397 | 1.25× |
| *F3 data-dependent per-block normalization* | | | | |
| F3-1 | `r47_dd_relu_d8` | – | – | – |
| F3-2 | `r61c_dd_meansq3_p512_d8` | 6838 | 6594 | 1.04× |
| F3-3 | `r61_dd_meansq_p512_d8` | 6838 | 6582 | 1.04× |
| F3-4 | `r61b_dd_absmean2_p768_d8` | 7942 | 6730 | 1.18× |
| F3-5 | `r47_dd_meanabs_d8` | – | – | – |
| F3-6 | `r61b_dd_absmean_p512_d8` | 7936 | 6732 | 1.18× |
| *F4 global-scalar coupling* | | | | |
| F4-1 | `r63_gsum_b0p02_p2048_d8` | 6614 | 5161 | 1.28× |
| F4-2 | `r63_gmean_b1p0_p2048_d8` | 6614 | 5169 | 1.28× |

## B END-TO-END TRAINING DETAILS

We instantiate one emitted family rewrite as the tensor-parallel collective of a 9.7-billion-parameter transformer (48 layers, model dimension 4096, 64 heads, sequence length 128) and train it for 120 steps (20 warm-up, 100 timed) on all 224 ranks at TP$=32\times$DP$=7$. The GPT-3-style and Llama-style architectures differ in their normalization and positional scheme; both use synthetic tokens,

Table 3: Production problem set, families F5–F8 — median-seed simulator time ($\mu$s); same configuration as Table 2.

| ID | log name | sim OverlayCCL ($\mu$s) | sim SorcarCCL ($\mu$s) | sim ratio |
|---|---|---|---|---|
| *F5 multiplicative global-norm clipping* | | | | |
| F5-1 | r65b_gnorm_b0p6_p2048_d8 | 6476 | 5387 | 1.20× |
| F5-2 | r65c_gnorm_b0p8_p2048_d8 | 6259 | 5387 | 1.16× |
| F5-3 | r65_gnorm_b2p0_p2048_d8 | 6259 | 5387 | 1.16× |
| F5-4 | r65b_gnorm_b0p7_p2048_d8 | 6629 | 5387 | 1.23× |
| F5-5 | r65_gnorm_b0p5_p2048_d8 | 6629 | 5387 | 1.23× |
| F5-6 | r65_gnorm_b0p5_p1024_d8 | 6244 | 5381 | 1.16× |
| F5-7 | r65b_gnorm_b0p5_p512_d8 | 6599 | 5382 | 1.23× |
| F5-8 | r65b_gnorm_b3p0_p2048_d8 | 6381 | 5387 | 1.18× |
| F5-9 | r65b_gnorm_b0p3_p2048_d8 | 6133 | 5387 | 1.14× |
| F5-10 | r65b_gnorm_b1p5_p2048_d8 | 6133 | 5387 | 1.14× |
| *F6 rank-1 parameter coupling* | | | | |
| F6-1 | r67b_vself_b0p4_p2048_d8 | 6643 | 5371 | 1.24× |
| F6-2 | r82_vself_b0p30_p2048_d8 | 6643 | 5371 | 1.24× |
| F6-3 | r79_vself_b0p35_p2048_d8 | 6643 | 5371 | 1.24× |
| *F7 banded / bidiagonal neighbor coupling* | | | | |
| F7-1 | r72_bidiag_b5_Ld5_Lo2_w1_d8 | 6858 | 6515 | 1.05× |
| F7-2 | r72_bidiag_b5_Ld4_Lo2_w1_d8 | 6834 | 6743 | 1.01× |
| F7-3 | r72_bidiag_b5_Ld4_Lo2_w2_d8 | 6834 | 6551 | 1.04× |
| F7-4 | r72_bidiag_b6_Ld5_Lo2_w2_d8 | 6888 | 6815 | 1.01× |
| F7-5 | r72_bidiag_b5_Ld4_Lo2_w3_d8 | 6820 | 6758 | 1.01× |
| *F8 block-structured rotation* | | | | |
| F8-1 | r73_mono_b6_a5_L5_w4_d8 | 6709 | 6486 | 1.03× |

Table 4: Production problem set, families F1–F4 — single-seed warm-cache hardware time (ms) at 224 ranks (second of two back-to-back runs, 100 timed iterations after 20 warm-up iterations).

| ID | log name | RT OverlayCCL (ms) | RT SorcarCCL (ms) | RT OverlayCCL/SorcarCCL |
|---|---|---|---|---|
| *F1 weighted per-shard gradient reduction* | | | | |
| F1-1 | r71_wdiag_b6_L5_w1_d8 | 13.17 | 4.99 | 2.64× |
| F1-2 | r71_wdiag_b5_L3_w1_d8 | 10.83 | 7.92 | 1.37× |
| *F2 MoE routing / expert-assignment coverage* | | | | |
| F2-1 | r60d_b16_L5_p2048_d8 | 17.57 | 4.33 | 4.06× |
| F2-2 | r60d_b10_L5_p2048_d8 | 16.67 | 4.29 | 3.89× |
| F2-3 | r60c_b12_L5_p2048_d8 | 15.31 | 4.09 | 3.74× |
| F2-4 | r60d_b13_L4_p2048_d8 | 15.30 | 4.19 | 3.65× |
| F2-5 | r43_route_B9_d8_L3_count8 | 12.20 | 4.53 | 2.69× |
| F2-6 | r40_route_d8_L3_count8 | 11.37 | 4.29 | 2.65× |
| F2-7 | r60c_b8_L2_p2048_d8 | 10.88 | 4.32 | 2.52× |
| F2-8 | r60c_b10_L4_p2048_d8 | 8.39 | 5.03 | 1.67× |
| F2-9 | r60_b5_L2_d8 | 8.90 | 5.37 | 1.66× |
| F2-10 | r40_route_d6_L3_count8 | 8.63 | 5.48 | 1.57× |
| F2-11 | r60c_b9_L3_p2048_d8 | 12.47 | 8.55 | 1.46× |
| *F3 data-dependent per-block normalization* | | | | |
| F3-1 | r47_dd_relu_d8 | 631.99 | 5.40 | 117.04× |
| F3-2 | r61c_dd_meansq3_p512_d8 | 24.05 | 5.50 | 4.37× |
| F3-3 | r61_dd_meansq_p512_d8 | 21.95 | 5.09 | 4.31× |
| F3-4 | r61b_dd_absmean2_p768_d8 | 24.36 | 5.82 | 4.19× |
| F3-5 | r47_dd_meanabs_d8 | 21.43 | 5.35 | 4.01× |
| F3-6 | r61b_dd_absmean_p512_d8 | 21.34 | 5.54 | 3.85× |
| *F4 global-scalar coupling* | | | | |
| F4-1 | r63_gsum_b0p02_p2048_d8 | 5.23 | 4.01 | 1.30× |
| F4-2 | r63_gmean_b1p0_p2048_d8 | 5.24 | 4.15 | 1.26× |

Table 5: Production problem set, families F5–F8 — single-seed warm-cache hardware time (ms); same configuration as Table 4.

| ID | log name | RT OverlayCCL (ms) | RT SorcarCCL (ms) | RT OverlayCCL/SorcarCCL |
|---|---|---|---|---|
| *F5 multiplicative global-norm clipping* | | | | |
| F5-1 | `r65b_gnorm_b0p6_p2048_d8` | 6.33 | 4.30 | 1.47× |
| F5-2 | `r65c_gnorm_b0p8_p2048_d8` | 5.89 | 4.06 | 1.45× |
| F5-3 | `r65_gnorm_b2p0_p2048_d8` | 5.98 | 4.17 | 1.43× |
| F5-4 | `r65b_gnorm_b0p7_p2048_d8` | 5.84 | 4.12 | 1.42× |
| F5-5 | `r65_gnorm_b0p5_p2048_d8` | 5.80 | 4.24 | 1.37× |
| F5-6 | `r65_gnorm_b0p5_p1024_d8` | 5.92 | 4.43 | 1.34× |
| F5-7 | `r65b_gnorm_b0p5_p512_d8` | 5.58 | 4.27 | 1.31× |
| F5-8 | `r65b_gnorm_b3p0_p2048_d8` | 5.21 | 4.39 | 1.19× |
| F5-9 | `r65b_gnorm_b0p3_p2048_d8` | 4.76 | 4.16 | 1.14× |
| F5-10 | `r65b_gnorm_b1p5_p2048_d8` | 4.86 | 4.35 | 1.12× |
| *F6 rank-1 parameter coupling* | | | | |
| F6-1 | `r67b_vself_b0p4_p2048_d8` | 9.09 | 4.23 | 2.15× |
| F6-2 | `r82_vself_b0p30_p2048_d8` | 18.93 | 16.52 | 1.15× |
| F6-3 | `r79_vself_b0p35_p2048_d8` | 18.60 | 16.58 | 1.12× |
| *F7 banded / bidiagonal neighbor coupling* | | | | |
| F7-1 | `r72_bidiag_b5_Ld5_Lo2_w1_d8` | 19.43 | 12.06 | 1.61× |
| F7-2 | `r72_bidiag_b5_Ld4_Lo2_w1_d8` | 17.82 | 13.42 | 1.33× |
| F7-3 | `r72_bidiag_b5_Ld4_Lo2_w2_d8` | 17.89 | 14.70 | 1.22× |
| F7-4 | `r72_bidiag_b6_Ld5_Lo2_w2_d8` | 20.53 | 17.66 | 1.16× |
| F7-5 | `r72_bidiag_b5_Ld4_Lo2_w3_d8` | 17.81 | 15.85 | 1.12× |
| *F8 block-structured rotation* | | | | |
| F8-1 | `r73_mono_b6_a5_L5_w4_d8` | 13.50 | 12.08 | 1.12× |

since the claim is throughput with loss equivalence, not convergence. Each backend loads its emitted kernel verbatim from the search output; the OverlayCCL and SorcarCCL runs share every other component (optimizer, initialization, data order), so any loss difference would come only from the collective schedule. The warm-cache step-time median over the 100 timed steps is in Table 6.

Table 6: End-to-end training at 224 ranks: warm-cache median step time over 100 timed steps for the OverlayCCL and SorcarCCL backends, the resulting step speedup, the in-situ family-collective speedup, and the final training loss (bit-identical across backends).

| architecture | OverlayCCL (ms) | SorcarCCL (ms) | step speedup | family speedup | final loss |
|---|---|---|---|---|---|
| Llama-style | 9386.3 | 4416.6 | 2.13× | 2.96× | 2.5496 |
| GPT-3-style | 9440.2 | 4396.5 | 2.15× | 2.94× | 2.8093 |

## C METHODOLOGY ABLATION

We disable one stage of the SorcarCCL research loop at a time and re-run the search on the 40-problem set, holding the simulator, proposer model, and budgets fixed. The complete loop (FULL) is compared against four ablations: removing idea proposal and pairwise judging (NOA), candidate selection and iterative refinement (NOB), the convergence check (NOC), and adversarial testing (NOADV). Each of the five arms searches all 40 problems under three seeds, for 600 runs in total.

On 55–65% of the problems every arm converges to the same program as FULL: the proposer solves the easy cases in a single pass regardless of which stage is removed. The methodology's value concentrates in a tail of harder problems on which an ablated arm fails to find the rewrite and reverts toward the baseline. We classify each (arm, problem) outcome against FULL's median-seed simulator time with a symmetric $\pm 20\%$ band (a tie within the band, a win above it, a loss below it) and report $\text{net} = \text{losses} - \text{wins}$ in Table 7. Every arm is net-negative and the ordering is monotone: idea proposal (NOA) and iterative refinement (NOB) are the most load-bearing, followed by adversarial testing and the convergence check. The wins are the symmetric mirror of the losses—seed noise in which an ablated arm's lucky seed reaches an optimum that FULL also finds on a sibling instance—so no ablation discovers a rewrite the full loop cannot.

Reading the emitted median-seed code confirms that the regressions are structural rather than noise: on the tail problems FULL recognizes that an iterated-collective loop collapses algebraically to two

Table 7: Methodology ablation on the 40 problems; **net** = losses − wins (higher is worse).

| arm | stage removed | tie | win | loss | net |
|---|---|---|---|---|---|
| NoA | propose & judge ideas | 31 | 1 | 8 | +7 |
| NoB | select & iterate | 31 | 1 | 8 | +7 |
| NoAdv | adversarial testing | 32 | 1 | 7 | +6 |
| NoC | convergence check | 33 | 1 | 6 | +5 |

**Algorithm 1** The Phase-2 research loop for one collective problem. Sim is the Phase-1 simulator; $s(\cdot)$ is `score_candidate`, which returns $\mathrm{Sim}(c)$ on a candidate that passes the correctness oracle $\mathcal{O}$ and an error class otherwise; $B_{\text{step}}$ and $B_{\$}$ are the step and dollar budgets.

**input:** formula $f$, baseline code $c_0$, reference document $R$
$c^\star \leftarrow c_0$; $v^\star \leftarrow s(c_0)$; ledger $\mathcal{L} \leftarrow \{(c_0, v^\star)\}$
**while** step budget $B_{\text{step}}$ and dollar budget $B_{\$}$ remain **and** goal not met **do**
  $I \leftarrow \textsc{ProposeIdeas}(f, R, \mathcal{L})$ ▷ rewrites excluding everything in $\mathcal{L}$
  $i^\star \leftarrow \arg\max_{i \in I} \textsc{JudgePairwise}(i)$ ▷ pick one idea
  $c \leftarrow \textsc{Implement}(i^\star)$
  $c \leftarrow \textsc{AdversarialFix}(c, \{x : \text{rank-asymmetric, non-uniform shape, op corner}\})$
  $v \leftarrow s(c)$ ▷ $v = \mathrm{Sim}(c)$ if $\mathcal{O}(c)$ holds, else an error class
  $\mathcal{L} \leftarrow \mathcal{L} \cup \{(c, v)\}$
  **if** $\mathcal{O}(c)$ **and** $v < v^\star$ **then** $c^\star \leftarrow c$; $v^\star \leftarrow v$
  **end if**
**end while**
**return** $c^\star$ ▷ gated at the training world size in Phase 3

all-reduces, whereas the ablated arm emits the literal loop with one collective per iteration (about seven collectives, matching the baseline), and the collapse becomes harder to find as loop depth grows. Algorithm 1 gives the complete Phase-2 loop that these arms ablate.

## D Divergence by family: OverlayCCL vs. SorcarCCL

For every family we show one representative problem: the median-seed program emitted by OverlayCCL and by SorcarCCL, condensed to the collective structure and the code that surrounds it (helper names shortened, correctness checks and comments elided). The two mechanisms recur throughout the pool. In F1, F2, F4, F5, F6, and F7 an algebraic identity lets SorcarCCL reduce the *number* of collectives; in F3 and F8 the collective count is already minimal and SorcarCCL instead *vectorizes* the per-block computation between collectives. In both cases OverlayCCL's strategy enumeration reaches a locally reasonable program—one collective per sub-step, or a correct but scalar per-block loop—and its refinement operators never rewrite the intervening computation, so it stops at that local optimum. The research loop diverges because it forms and tests hypotheses about the computation *between* collectives rather than choosing among predefined collective strategies.

### D.1 F1 — weighted per-shard reduction

Here both controllers issue the same number of all-reduces; the gap is that OverlayCCL rebuilds the masked, scaled buffer with a per-block Python slice loop each round, which lowers to scalar-indexed slice writes, while SorcarCCL precomputes the weight and normalization vectors once and applies them as whole-tensor products.

```
s = all_reduce(x)
for step in range(L):
    buf = s.clone()
    for b in range(B):                # per-block Python slice loop
        buf[b*S:(b+1)*S] = w[step] * s[b*S:(b+1)*S]
    acc = all_reduce(buf)
    for b in range(B):
        s[b*S:(b+1)*S] = acc[b*S:(b+1)*S] / A[b]
return s
```

Listing 1: F1 (`r71_wdiag_b6_L5`) — OverlayCCL

```
s = all_reduce(x)
weighted_mask = torch.cat([w[k].expand(S) for k in ...])  # built once
norm_inv      = torch.cat([(1.0 / A[b]).expand(S) for b in range(B)])
for step in range(L):
    acc = all_reduce(s * weighted_mask)   # whole-tensor product, no loop
    s = acc * norm_inv
return s
```

Listing 2: F1 (`r71_wdiag_b6_L5`) — SorcarCCL

### D.2 F2 — MIXTURE-OF-EXPERTS ROUTING COVERAGE

This is the case study of Section 4.3. OverlayCCL keeps one masked all-reduce per round and renormalizes each block by its coverage count $c_b$; because $c_b$ is a constant, the whole cascade of eight all-reduces collapses to two.

```
s = all_reduce(x)
for step in range(L):                 # one masked all-reduce per round
    buf = s * keep_mask[step]         # mask this round's covered blocks
    acc = all_reduce(buf)
    for b in range(B):
        acc[b*S:(b+1)*S] /= c[b]      # renormalize by coverage count
    s = acc
return all_reduce(s)                  # closing reduction
```

Listing 3: F2-1 (`r60d_b16_L5`) — OverlayCCL: 8 all-reduces

```
s = all_reduce(x)                     # c[b] are constants => the masked
mask = build_coverage_mask(c)         # renormalize-reduce recurrence
s = all_reduce(s * mask)              # telescopes to a single masked AR
return s
```

Listing 4: F2-1 (`r60d_b16_L5`) — SorcarCCL: 2 all-reduces

### D.3 F3 — PER-BLOCK NORMALIZATION

This is the complementary vectorization example discussed in Section 4.3. The collectives are irreducible; the gap is that OverlayCCL computes the data-dependent per-block factor $f_b$ in a scalar loop, whereas SorcarCCL reshapes to $(B, S)$ and reduces along the block axis in one vectorized op.

```
s = all_reduce(x)
for step in range(L):
    buf = s.clone()
    for b in range(B):                          # per-block scalar loop
        sb = s[b*S:(b+1)*S]
        f  = 1.0 + 3.0 * (sb * sb).mean()       # data-dependent factor
        buf[b*S:(b+1)*S] = sb * f
    s = all_reduce(buf) / world_size
    for b in range(B):
        s[b*S:(b+1)*S] /= (1.0 + 3.0 * ...)
return s
```

Listing 5: F3-3 (`r61_dd_meansq`) — OverlayCCL

```
s = all_reduce(x).view(B, S)
for step in range(L):
    f = (1.0 + 3.0 * (s * s).mean(dim=1)).unsqueeze(1)  # all blocks at once
    s = all_reduce((s * f).view(-1)).view(B, S) / (world_size * f)
return s.view(-1)
```

Listing 6: F3-3 (`r61_dd_meansq`) — SorcarCCL

### D.4 F4 — GLOBAL-SCALAR COUPLING

OverlayCCL treats the scalar-sum coupling and the payload reduction as separate collectives repeated per sub-step; SorcarCCL observes that the coupling is linear in the reduced payload and folds everything into one all-reduce plus a local scalar correction.

```
s = all_reduce(x)
for step in range(L):
    buf = s + BETA * s.sum()          # couple to the global sum
    acc = all_reduce(buf) / world_size
    s = acc - 0.02 * acc.sum() / (1.0 + 0.02 * D)
return s
```

Listing 7: F4 (`r63_gsum_b0p02`) — OverlayCCL

```
s = all_reduce(x)                     # linear coupling folds:
return s + 0.02 * s.sum()             # one all-reduce, local correction
```

Listing 8: F4 (`r63_gsum_b0p02`) — SorcarCCL

### D.5 F5 — GLOBAL-NORM CLIPPING

OverlayCCL stacks the squared-norm and payload reductions and re-derives the clip factor each sub-step (its emitted trajectory visibly flails between several stacking layouts); SorcarCCL computes the norm factor from a single reduced payload and applies it before one final reduction.

```
s = all_reduce(x)
for step in range(L):
    stacked = torch.stack([s, s.abs()])   # batch payload + norm term
    red = all_reduce(stacked)
    norm = 1.0 + 0.6 * red[1].mean() / world_size
    s = red[0] / (world_size * norm)
return s
```

Listing 9: F5 (`r65b_gnorm_b0p6`) — OverlayCCL

```
s = all_reduce(x)
norm_factor = (1.0 + 0.6 * s.abs().mean()) * world_size
return all_reduce(s / norm_factor)
```

Listing 10: F5 (`r65b_gnorm_b0p6`) — SorcarCCL: 2 all-reduces

### D.6 F6 — RANK-1 COUPLING

The update couples the payload to a rank-1 term $v\,v^\top s$. OverlayCCL reduces the full coupled buffer once per sub-step; SorcarCCL reduces the payload once, reconstructs the rank-1 term locally, and reduces the corrected buffer, cutting the collective count.

```
s = all_reduce(x)
for step in range(L):
    v = (1 - 2 * (idx % 2)).float()          # +/-1 pattern
    buf = s + BETA * v * (v * s).mean()       # rank-1 coupling
    s = all_reduce(buf) / world_size
return s
```

Listing 11: F6 (`r67b_vself_b0p4`) — OverlayCCL

```
s = all_reduce(x)
v = (1 - 2 * (idx % 2)).float()
buf = s + BETA * v * (v * s).mean()          # reconstruct locally
s = all_reduce(buf) / world_size
return s
```

Listing 12: F6 (`r67b_vself_b0p4`) — SorcarCCL: 2 all-reduces

### D.7 F7 — BANDED NEIGHBOR COUPLING

OverlayCCL fully unrolls the banded solve, rebuilding the diagonal/off-diagonal recurrence and back-substitution block by block each round; SorcarCCL precomputes the recurrence coefficients once and vectorizes the diagonal and off-diagonal application across blocks, letting the band compose into a single reduction over the combined payload.

```
s = all_reduce(x)
for step in range(L):
    for b in range(B):                    # rebuild recurrence per block
        A[b] = wd + (Cprev if b else 0.0)
        buf[b*S:(b+1)*S] = A[b] * s[b*S:(b+1)*S] + off * s_prev
```

```
    acc = all_reduce(buf)
    for b in reversed(range(B)):           # back-substitution
        s[b*S:(b+1)*S] = solve(acc, b)
return s
```

Listing 13: F7 (`r72_bidiag_b5`) — OverlayCCL

```
A, C = precompute_recurrence(wd, off, B)   # once, outside the loop
for step in range(L):
    diag = A * s                           # vectorized diagonal
    offd = C * shift(s)                    # vectorized off-diagonal
    s = all_reduce((diag + offd).view(-1)).view(B, S)
return s.view(-1)
```

Listing 14: F7 (`r72_bidiag_b5`) — SorcarCCL

### D.8 F8 — BLOCK ROTATION

The collectives here are already few; the gap is marshalling. OverlayCCL applies each block's $2 \times 2$ rotation in a per-block loop and batches sub-steps into stacked groups; SorcarCCL expresses the rotation as a single vectorized permute-and-scale over all blocks.

```
groups = [[0,1,2], [3,4,5], [6]]           # 3 stacked collectives
for g in groups:
    multi_buf = torch.stack([rotate_blocks(s, step) for step in g])
    red = all_reduce(multi_buf)
    for j, step in enumerate(g):
        for b in range(B):                 # per-block rotation apply
            s[b*S:(b+1)*S] = red[j, b*S:(b+1)*S] / A[b]
return s
```

Listing 15: F8 (`r73_mono_b6_a5`) — OverlayCCL

```
perm = build_rotation_perm(B)              # vectorized permute + scale
for step in range(L):
    buf = s[perm] * sig                    # apply all block rotations at once
    s = all_reduce(buf) / A_vec
return s
```

Listing 16: F8 (`r73_mono_b6_a5`) — SorcarCCL