ParallelKernelBench: Can LLMs Write Fast Multi-GPU Kernels?

Anonymous Authors[1]

Abstract

There is growing interest in using large language models (LLMs) to write high-performance GPU kernels, with recent work showing promising results on single-GPU workloads. However, multi-GPU kernel generation remains unexplored, despite communication emerging as a dominant bottleneck in large-scale training and inference. In this paper, we study how well LLMs can write multi-GPU kernels, a task compounded by three challenges: (1) the design space is combinatorially large, as training and inference workloads can be parallelized across tensor, expert, pipeline, data, and sequence dimensions; (2) single-GPU memory-compute roofline analysis fails to capture communication bottlenecks in multi-GPU execution; and (3) the many hardware paths available for communication (e.g., copy engine, TMA, SM instructions) each carry distinct tradeoffs. We introduce **ParallelKernelBench** (PKB), a benchmark and evaluation framework for multi-GPU kernel generation. Additionally, we construct a taxonomy of distributed workloads spanning different parallelism types and select 87 problems covering compositions that arise in real workloads. We contribute evaluations of frontier coding models on PKB, finding that current LLMs struggle: single-shot kernel correctness plateaus at 32% of cases, with speedup exceeding an unoverlapped baseline (PyTorch + NCCL) in only 25% of cases. We also contribute a communication-aware roofline analysis of correct kernels, finding that over 90% of baselines achieve less than 50% of peak hardware utilization. Optimized solutions to many PKB workloads are largely absent from existing open-source repositories; we highlight several LLM-generated net new kernels that outperform their reference implementations in specific regimes, including NeMo's vocab-parallel log-probability kernel (up to $2.06\times$), Hyena CP (up to $1.72\times$), and SAM3 IoU suppression (up to $1.40\times$).

1. Introduction

In recent years, large language models (LLMs) have made rapid progress on GPU kernel generation (Lange et al., 2025; Baronio et al., 2025). Several benchmarks have emerged to track and drive this progress, establishing standardized evaluation frameworks that guide model development to optimize single-GPU workloads (Ouyang et al., 2025; Lin et al., 2026; Xing et al., 2026). As a result, the majority of efforts to automate kernel generation, including agentic scaffolding (Dai et al., 2026), model finetuning (Baronio et al., 2025), and emerging industry tools (Kernel, 2026) have focused on the single-GPU regime. However, modern AI workloads increasingly require kernels that span multiple GPUs due to increasing scale, yet the ability of LLMs to generate correct and efficient multi-GPU kernels remains largely unexplored; as of writing, no standardized benchmark or evaluation exists to understand model capabilities in this setting.

Frontier LLM architectures have outgrown the memory and compute of a single GPU, introducing new bottlenecks (Anthony et al., 2024). On many production distributed training and inference workloads, communication increasingly consumes the majority of runtime and yields low Model FLOPs Utilization at scale (Duan et al., 2024; Jiang et al., 2024; Fernandez et al., 2025); for Llama-3.3-70B (Grattafiori et al., 2024), all-reduce alone accounts for over 20% of end-to-end inference latency on a DGX H100, even with NVLink and in-switch reductions (Gond et al., 2025). Hardware evolution has widened this gap: FP16 tensor core throughput on Streaming Multiprocessors (SMs) has grown over $10\times$ while NVLink bandwidth has grown only $6\times$ from NVIDIA Ampere (Krashinsky et al., 2023) to Rubin (Mitrasish, 2026), and scale-up domains such as NVL144 and NVL576 continue to add complexity (Bhargava et al., 2026). While accelerators have converged on technologies like HBM and systolic arrays for GEMMs, the networking landscape remains heterogeneous—NVIDIA uses NVLink, AMD uses

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

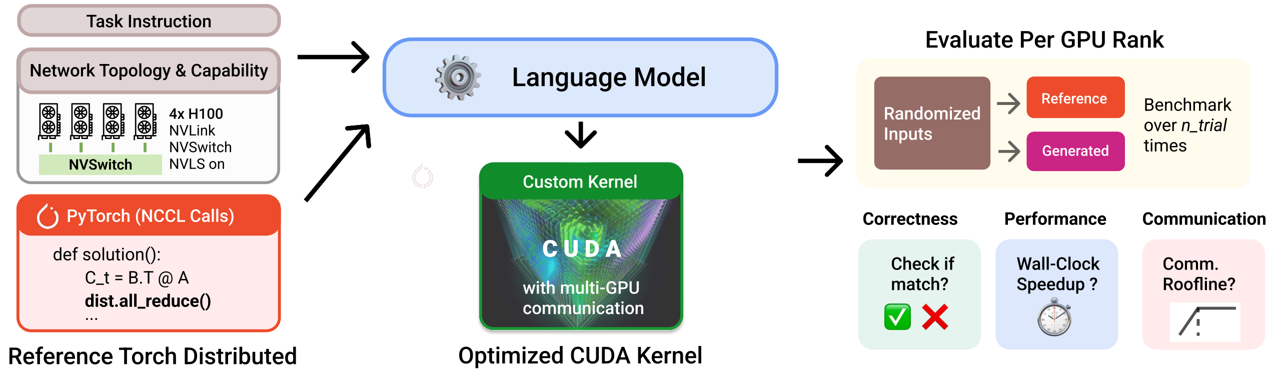


Figure 1. **ParallelKernelBench evaluates language models on multi-GPU kernel optimization.** Given a system topology and an unoverlapped PyTorch/NCCL reference implementation, models must rewrite the program into a custom distributed CUDA kernel using device-side communication primitives. The reference uses NCCL collectives, the dominant communication library in modern distributed training frameworks, while the target solution replaces these with fine-grained CUDA kernels that exploit Unified Virtual Addressing (UVA) for direct peer-to-peer transfers over NVLink.

xGMI (Schieffer et al., 2026), TPUs use Inter-Chip Interconnect (Jouppi et al., 2023)—with new topologies still emerging. This diversity makes a parallel kernel benchmark all the more valuable and timely.

Evaluating LLMs on multi-GPU kernel generation, however, presents three distinct challenges:

1. The **problem design space** expands combinatorially beyond the single-GPU case. Practitioners compose parallelism strategies (e.g. Tensor, Expert, Data, Context, and Sequence Parallelism) according to hardware constraints (Narayanan et al., 2021; Grattafiori et al., 2024), with each composition inducing a different communication pattern. This results in a vast space of problems that a benchmark must cover, and it is unclear which combinations are characteristic of AI systems encountered in practice.

2. Existing evaluation frameworks are **not communication-aware**. Standard evaluation often compares a kernel against the compute and memory-bandwidth limits predicted by the standard Roofline model (Williams et al., 2009; Czaja et al., 2020). In distributed GPU programs, however, the relevant bottleneck might include interconnect bandwidth, collective latency, or taxonomy-dependent communication rather than local compute or memory bandwidth. Thus, even defining the right performance target requires per-problem cost models that account for hardware topology (Checconi et al., 2025; Cardwell & Song, 2019).

3. Unlike single-GPU kernels, which simply load data, compute, and store results within a single kernel, multi-GPU kernels introduce a new axis of **communication mechanism selection**. Communication can be placed intra-SM (fused into the same kernel as compute) or inter-SM (in a separate kernel), and launched via different hardware mechanisms (e.g., copy engine, TMA, SM load/store, NVLS), each with distinct performance tradeoffs (Sul et al., 2025).

To address these challenges, we introduce **ParallelKernelBench** (PKB), a benchmark designed to generate and evaluate multi-GPU kernels. PKB provides 87 problems organized under a taxonomy that spans different parallelism strategies and distills core communication patterns that arise in real AI workloads, from inference to RL to post-training. The benchmark takes a naive PyTorch and NCCL reference as input and challenges models to produce an optimized CUDA kernel, directly exposing model capabilities on low-level distributed optimization. To evaluate and analyze generated kernels, we propose a communication-aware roofline model that extends the standard roofline with interconnect bandwidth: situating both baseline and generated implementations against hardware limits across *compute, memory, and communication* bottlenecks.

We evaluate frontier coding models (GPT-5.5, Opus-4.7, Gemini 3 Pro, GLM-5.1, DeepSeek V4 Pro) on both correctness and performance. We note models perform poorly: with GPT-5.5, fewer than 33% of generated kernels are correct, and only 25% match or exceed the unoverlapped Torch + NCCL baseline (fast_1). The failures are attributed to challenging syntax, complex reasoning of parallelization, and underutilization of advanced communication hardware

110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164

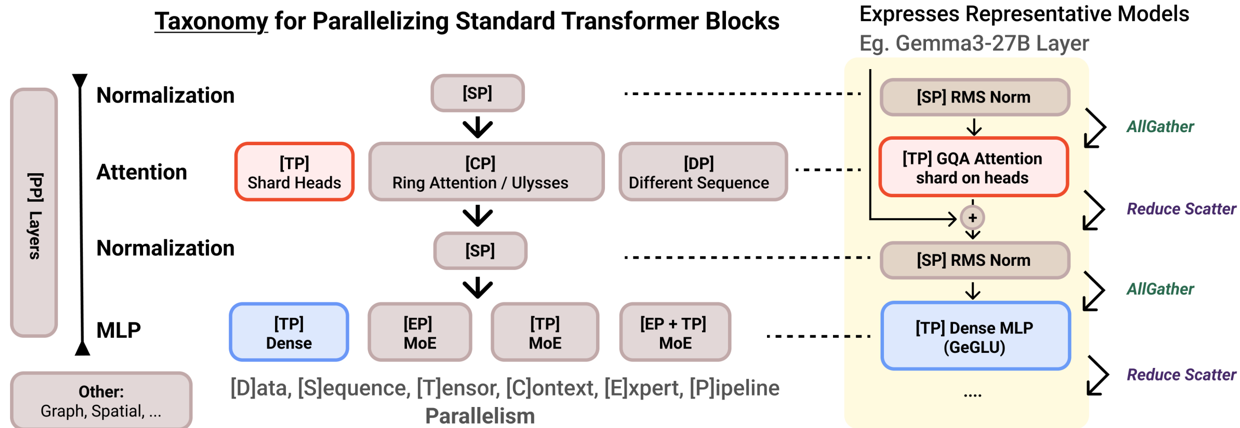


Figure 2. **A taxonomy of parallelism strategies for standard transformer workloads.** A standard transformer layer (norm → attention → norm → MLP) can be parallelized across data, sequence, tensor, context, layer/pipeline, and expert dimensions. We provide an instantiation of this taxonomy as an example on a `Gemma3-27B` layer, annotating each operation with its corresponding NCCL collective. This taxonomy guides the construction of PKB problems.

features. Using an agentic harness (Yang et al., 2024) with execution feedback raises correctness to 36.7% and fast_1 to 29.8%; iterative refinement helps, but substantial headroom remains. Interestingly, we note open-source models underperform significantly, notable given their rapid progress on single-GPU benchmarks (e.g. GLM-5.1 (GLM-5-Team et al., 2026) highlights matching Opus on KernelBench Level 3). Despite these limitations, we note that generated solutions at times produce *net-new* high-performance kernels, including NeMo vocab-parallel log-probability (up to 2.06×), Hyena CP (up to 1.72×), and SAM3 IoU suppression (up to 1.40×)—as well as speedups on widely-used operations such as Ulysses attention (up to 1.83×); see Appendix G for details.

These findings highlight the gap between current model capabilities and fully autonomous LLM-based optimization of multi-GPU workloads. We release PKB as an open benchmark environment to drive community progress toward automated distributed GPU code generation. Concretely, we contribute:

1. A **taxonomy** for describing distributed AI workloads in terms of parallelism strategies and sharding configurations, used to organize and curate PKB's problem set.

2. ParallelKernelBench, a **benchmark** of 87 diverse, real-world, and *net new* multi-GPU problems paired with a communication-aware evaluation framework that situates kernel performance against compute, memory, and communication ceilings.

3. **Analysis** of kernels generated by frontier LLMs, revealing that models systematically underperform by defaulting to simple communication primitives rather than more complex, higher-performing alternatives.

2. Background & Related Work

In this section, we provide background on modern GPU networking hardware and software, as well as a review of prior work (Table 1) related to LLM-driven kernel generation.

2.1. Background

GPU Architecture. Modern datacenter-grade NVIDIA GPUs execute programs called *kernels* across thousands of parallel hardware threads organized into a three-level hierarchy: *threads* are the basic unit of execution; threads are grouped into *warps* of 32 that execute on a single SM; and warps are grouped into *blocks* that share a fast on-chip shared memory scratchpad. GPUs also expose high-bandwidth memory (HBM) as the primary large working set memory. On multi-GPU systems, GPUs are connected via *NVLink*, an interconnect that allows threads executing on one GPU to directly read from and write to the HBM of a peer GPU.

Interconnect Hardware. Modern GPU clusters expose a hierarchy of interconnects. PCIe (64 GB/s) handles CPU-to-GPU transfers, while NVLink (450 GB/s unidirectional per GPU) provides high-bandwidth point-to-point GPU-to-GPU connectivity within a node. NVSwitch connects all GPUs within a node in a full all-to-all fabric and supports in-network compute via NVLink SHARP (NVLS)—exposed

Benchmark	Qty.	M-GPU	Workload	Sources
KernelBench	250	×	Inf.	PyTorch/HF
FlashInfer-B	18	×	Inf.	Serving Traces
SOL-ExecB	235	×	Inf.	Subgraphs
MultiKernelB	285	×	Tr./Inf.	KB Ext.
PKB	**87**	✓	Tr./Inf.	Prod. Libs

Table 1. **Benchmark Comparison.** ParallelKernelBench (PKB) is the unique suite supporting distributed multi-GPU optimization. Table abbreviations: **M-GPU** (Multi-GPU), **Qty.** (Number of Problems), **Inf.** (Inference), **Tr.** (Training), **HF** (HuggingFace), and **Prod. Libs** (Production Libraries).

through `multimem` instructions—accelerating multicast and reduction operations without routing data through GPU SMs. Unless otherwise noted, all inter-GPU communication in this paper occurs over NVLink/NVSwitch.

Transfer Mechanisms. Following the taxonomy of ParallelKittens (Sul et al., 2025), GPU-to-GPU data transfer occurs via three primary mechanisms. The *copy engine* is host-initiated and saturates NVLink bandwidth for large messages, but cannot be invoked from within a kernel. *Tensor Memory Accelerator (TMA)* transfers move data asynchronously without consuming SM resources, making them well-suited for overlapping communication with compute. Finally, *SM load/store (LSA)* instructions perform transfers directly using SMs; while this increases register pressure, it enables NVLS in-switch compute via NVSwitch, which the other two mechanisms do not support.

Communication Software. NCCL is the standard communication library for GPU collectives and is used by `torch.distributed`: exposing operations such as all-reduce and reduce-scatter as asynchronous kernels enqueued on a CUDA stream (Hu et al., 2026b). While convenient, NCCL typically operates at the granularity of full collectives with limited flexibility for fine-grained compute-communication overlap. For lower-level control, programmers can use a *symmetric heap* - a memory region registered across all participating GPUs - to perform direct one-sided puts and gets from within a kernel (PyTorch Team, 2026). PyTorch exposes this via its symmetric memory abstraction, accessible from CUDA through Unified Virtual Addressing (UVA), which maps remote GPU memory into a single virtual address space so that kernels can read from and write to peer GPUs using standard load/store instructions.

2.2. Related Work

Custom Distributed Kernels. Many specialized libraries exist to optimize specific AI operations by overlapping communication with computation, such as Flux (Lin et al., 2025), DeepEP (DeepSeek-AI et al., 2025), and FlashMoE (Aimuyo et al., 2025). They employ fine-grained device-side communication operations to yield substantial latency and throughput gains over naive unoverlapped implementations. However, these libraries are manually tuned and do not generalize between hardware generations, motivating automated kernel generation.

LLM-Based Kernel Generation. KernelBench (Ouyang et al., 2025) established the standard benchmark for LLM-generated GPU kernels, with subsequent work (e.g. FlashInfer-Bench (Xing et al., 2026), SOL-ExecBench (Lin et al., 2026), MultiKernelBench (Wen et al., 2025)) expanding coverage to new problem types and hardware platforms. This has spurred optimization efforts including evolutionary search (Lange et al., 2025) and multi turn-based RL (Baronio et al., 2025) to improve kernel quality. However, all of these works operate exclusively in the single-GPU setting. The closest multi-GPU effort is CUCo (Hu et al., 2026a), an agentic framework for joint computation-communication kernel generation, but it evaluates only four workloads and restricts to the transformation of replacing host-initiated NCCL collectives with device-initiated NCCL GIN calls.

Multi-GPU Kernel DSLs. Several frameworks lower the barrier to writing multi-GPU kernels, such as ParallelKittens (Sul et al., 2025), Triton-Distributed (Zheng et al., 2025), Iris (Awad et al., 2025), and PyTorch Symmetric Memory (yifuwang et al., 2025). However, all still require expert human implementation. For a discussion of device-side communication primitives and a survey of modern kernel-writing frameworks, see Appendix B.

3. ParallelKernelBench

In this section, we describe ParallelKernelBench: its task format, the principles guiding problem selection, and its evaluation metrics.

3.1. ParallelKernelBench Task Format

Each benchmark task presents the model with an unoptimized reference implementation written in PyTorch with `torch.distributed` NCCL operations, along with a system topology specifying the number of ranks and intra-node hardware configuration. The model is tasked with rewriting this reference into a performant CUDA kernel that uses unified virtual addressing. This is performed via PyTorch's symmetric memory API, an interface for low-level peer-to-peer GPU communication in PyTorch. The end-to-end generation and evaluation cycle is illustrated in Figure 1 and the full prompt template is provided in Appendix A.

Prerequisites. Each task defines a case in the common `create_input_tensor()` function that initializes per-rank input tensors and sizes, and pre-configures all communication groups. The sharding scheme is fully determined before the model's `solution()` function is in-

voked, which runs identically on every rank. Note that depending on the task, per-rank inputs may differ. One minimal in-context example is provided: a distributed add kernel demonstrating symmetric memory setup/pointer caching and NVLink P2P data transfer via pointers.

Task input. For each task, the model is provided: (i) a reference implementation using unoptimized `torch.distributed` and NCCL collectives; (ii) a system prompt and in-context example; and (iii) a natural language system topology description specifying hardware capabilities such as NVLink P2P access and in-switch reduction support.

Task output. The model produces a lower-level rewrite of the reference `solution` function that replaces the unoptimized collective with a fine-grained kernel that overlaps computation and communication. The primary evaluation uses **CUDA** with **Torch Symmetric Memory** via PyTorch `load_inline`. Symmetric heap pointers are cached across trials so that setup overhead is excluded from measured latency.

3.2. Problem Selection

PKB problems are drawn from two signals of practical relevance: GitHub repositories scraped to identify operations frequent in real deployments, and optimized library implementations as evidence that the community considers a workload worth optimizing. Together, these sources ensure PKB reflects what practitioners encounter in production. We further characterize standard transformer models from these sources (Figure 2), decomposing each layer into parallelizable operations and annotating the required NCCL collectives; this serves as a systematic guide for constructing PKB problems with coverage across parallelism strategies and collective types. See Figure 3 for a survey of PKB's data sources and Appendix C for source links.

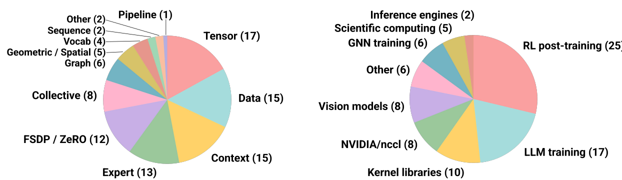
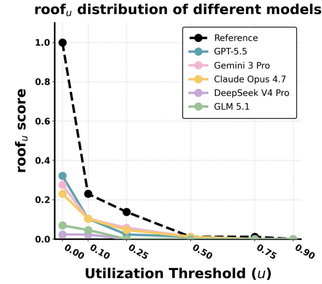


Figure 3. **ParallelKernelBench covers a diverse set of distributed GPU workloads.** Problems span the major parallelism strategies used in production and are sourced from industry codebases.



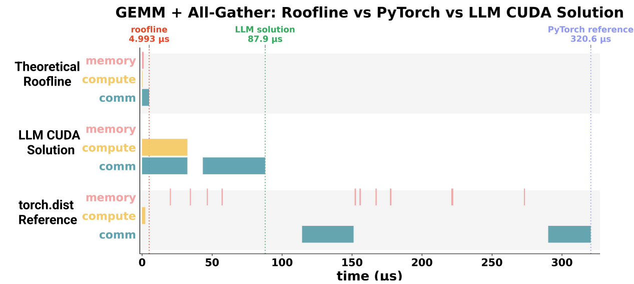


Figure 4. **PyTorch baselines fall far below the roofline; LLM solutions can close the gap.** (*Top*) Distribution of roof$_u$ scores shows both reference and generated kernels substantially underperform in hardware utilization. (*Bottom*) Profiler trace for problem `14_gemm_all_gather`: the PyTorch reference launches GEMM, NCCL all-gather, and a concat copy as separate kernels with stream-sync overhead between them. The optimized solution fuses GEMM and gather into a single WMMA kernel that writes output tiles directly to every peer's symmetric buffer over NVLink, decreasing runtime toward the theoretical minimum. Gaps are largely CPU scheduling latency.

4. Understanding the Baseline

We describe the metrics used to evaluate LLM-generated kernels and show that correct baseline solutions remain far from their theoretical hardware limits, establishing that high-performance solutions to PKB problems would be genuinely net new.

4.1. Metric Design

When comparing across models, we report $fast_p$ as introduced in KernelBench (Ouyang et al., 2025), which measures the proportion of tasks for which a model produces a kernel that is both correct and faster than the baseline. This provides a single scalar summary of model capability that jointly penalizes both incorrectness and failure to improve over the naive implementation. We also evaluate kernels along correctness, performance, and hardware utilization.

Correctness. For each task, we run 5 trials with random per-rank PyTorch inputs and verify that the output of the generated `solution()` matches that of the reference within numerical tolerance.

Performance. We measure wall-clock execution time averaged over 100 profiling iterations, preceded by 500 warmup iterations to reach a power-steady state, with kernels launched back-to-back without intermediate synchronization (Sul & Re, 2026). Generated kernels are compared against the unoverlapped PyTorch + NCCL reference baseline. Further details and variance between runs is reported in Appendix E.

4.2. Understanding the Communication Bottleneck

We introduce roof_u, the fraction of tasks for which the generated solution is both correct and achieves at least fraction u of the hardware-limited roofline bound. We contribute the first systematic application of roofline analysis as an automatic evaluation metric over a diverse distributed kernel benchmark; while fast_p asks whether a solution beats the PyTorch baseline, roof_u asks whether it uses the hardware well — measuring utilization relative to whichever resource (compute, memory, or communication) is the actual bottleneck.

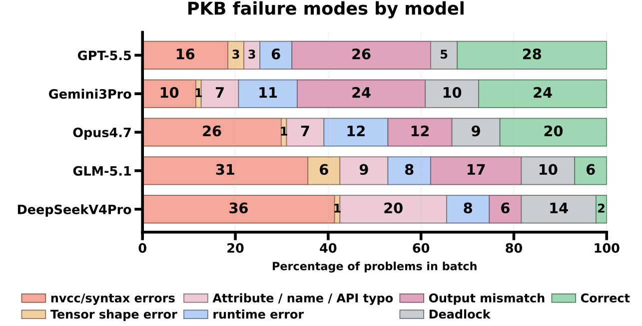
Concretely, for each task i we contribute a theoretical cost model (Appendix D) that derives roofline performance estimates from FLOP and byte counts, modeling the fastest achievable runtime under perfect overlap of compute, memory, and communication, i.e. the runtime if the solution were limited only by the binding hardware resource (Checconi et al., 2025).

$$T_{\mathrm{roof},i} := \max \left(\begin{array}{c} \dfrac{\mathrm{FLOPs}_i}{\text{Peak Compute Throughput}}, \\ \dfrac{\text{Bytes Moved From Memory}_i}{\text{Peak Memory Bandwidth}}, \\ \dfrac{\text{Bytes Communicated}_i}{\text{Peak NVLink Bandwidth}} \end{array} \right) \quad (1)$$

Roofline utilization is then $U_i = \min(1, T_{\mathrm{roof},i}/T_{\mathrm{sol},i})$, and

$$\mathrm{roof}_u := \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}\left[\mathrm{correct}_i \wedge U_i > u\right] \quad (2)$$

Baseline programs underutilize hardware. Before evaluating LLM-generated kernels, we first ask whether the naive PyTorch + NCCL reference leaves meaningful headroom — if baselines already approach hardware limits, there is little to optimize. Figure 4 shows the opposite: the majority of PyTorch baselines fall below 50% of their communication-aware roofline bound. See Appendix Table 5 for a per-bottleneck breakdown showing this gap is starkest on communication-bound tasks. With more fine-grained operations, LLM-generated solutions can potentially close this gap.



Comm. Mechanism Use by Model

Model	CE	LSA	NVLS	TMA
GPT-5.5	46.4%	100%	7.1%	0%
Opus-4.7	75%	100%	5%	0%
Gem.3 Pro	70.8%	100%	12.5%	0%
GLM-5.1	100%	100%	0%	0%
DSV4 Pro	50%	100%	0%	0%

Figure 5. **Single-shot LLMs struggle to parallelize computation correctly and default to simplistic inter-GPU transfer mechanisms.** Reasoning models reduce syntax errors, but correctness remains low, revealing a fundamental gap in understanding multi-GPU distributed memory and rank coordination. CE (copy engine) and LSA (SM load/store) dominate generated kernels; NVLS/`multimem` and TMA instructions for low-overhead data movement are nearly absent.

5. Model Evaluation

Models are evaluated via a single-attempt prompt containing an in-context example covering key boilerplate code, such as symmetric memory allocation, P2P data transfer, and using the `multimem` NVLS instruction (see Appendix A). All experiments were conducted on a single node with 4 NVIDIA H100 80GB GPUs interconnected via 4th-generation NVLink/NVSwitch running PyTorch 2.11 and CUDA 12.6. All inputs use BF16 precision.

5.1. Correctness & Error Analysis

Table 2 shows that correct solutions concentrate in collective primitives, TP-sharded GEMMs, and Ulysses context-parallel problems across all models. We hypothesize this reflects training data bias: these patterns are heavily represented in open-source multi-GPU frameworks (Sul et al., 2025; Sul & Re, 2026), and optimized reference implementations are widely available online (Hu et al., 2026b; Sul et al., 2025; Jacobs et al., 2023), giving models stronger priors.

Figure 5 classifies failure modes across all problems. Open-source and non-reasoning models fail predominantly at the syntax level — compiler and interpreter errors indicate they

Category	#	GPT-5.5		Claude Opus 4.7		Gemini 3 Pro		GLM-5.1		DeepSeek V4 Pro	
		pass @1→3	$fast_1$ @1→3	pass @1→3	$fast_1$ @1→3	pass @1→3	$fast_1$ @1→3	pass @1→3	$fast_1$ @1→3	pass @1→3	$fast_1$ @1→3
Collective Primitive	8	3 → 5	3 → 4	4 → 5	3 → 4	6 → 6	2 → 2	2 → 3	2 → 3	0 → 0	0 → 0
Tensor Parallel	17	2 → 2	1 → 2	1 → 3	1 → 3	3 → 3	1 → 1	1 → 1	0 → 0	0 → 0	0 → 0
Sequence Parallel	2	1 → 1	1 → 1	0 → 0	0 → 0	0 → 0	0 → 0	0 → 0	0 → 0	0 → 0	0 → 0
Context Parallel	12	7 → 8	5 → 6	7 → 7	3 → 4	7 → 9	5 → 7	2 → 2	0 → 0	2 → 3	0 → 1
Pipeline Parallel	1	0 → 0	0 → 0	0 → 0	0 → 0	0 → 0	0 → 0	0 → 0	0 → 0	0 → 0	0 → 0
Data Parallel	9	1 → 2	1 → 2	1 → 1	1 → 1	0 → 1	0 → 1	0 → 0	0 → 0	0 → 0	0 → 0
Expert Parallel	11	3 → 3	2 → 3	2 → 6	0 → 1	3 → 4	0 → 1	0 → 0	0 → 0	0 → 0	0 → 0
FSDP / ZeRO	9	3 → 4	3 → 4	3 → 3	3 → 3	1 → 1	0 → 1	1 → 1	0 → 0	0 → 0	0 → 0
Vocab Parallel	4	2 → 2	1 → 1	1 → 2	1 → 2	2 → 2	2 → 2	0 → 0	0 → 0	0 → 0	0 → 0
Graph Parallel	6	2 → 4	2 → 3	0 → 2	0 → 2	1 → 1	1 → 1	0 → 0	0 → 0	0 → 0	0 → 0
Geometric / Spatial	5	3 → 3	3 → 3	1 → 2	0 → 0	0 → 1	0 → 1	0 → 0	0 → 0	0 → 0	0 → 0
Other	2	1 → 2	0 → 1	0 → 0	0 → 0	1 → 2	1 → 2	0 → 0	0 → 0	0 → 0	0 → 0
Total	**87**	**28 → 36**	**22 → 27**	**20 → 31**	**12 → 20**	**24 → 30**	**12 → 19**	**6 → 7**	**2 → 3**	**2 → 3**	**0 → 1**

Table 2. **Single-shot LLMs struggle on PKB.** pass@k reports correctness on the best of k attempts; $fast_1$@k counts solutions that are both correct and outperform the PyTorch + NCCL baseline. Repeated sampling improves both metrics, but $fast_1$@3 still peaks at just 31% (GPT-5.5), suggesting fundamental limitations beyond sampling noise.

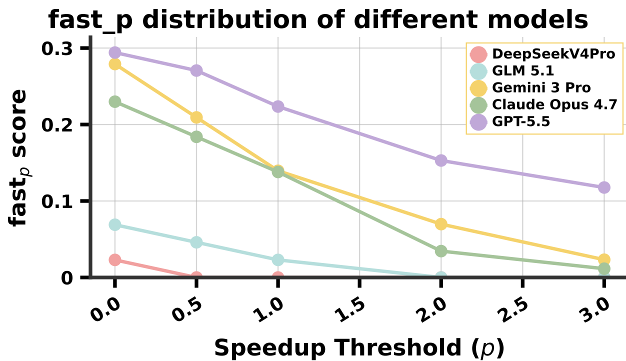


Figure 6. **Single-shot kernels rarely achieve meaningful speedup.** $fast_p$ (the percent of correct kernels exceeding p× speedup) drops sharply with p. Models exceed the unoverlapped baseline in fewer than 30% of cases.

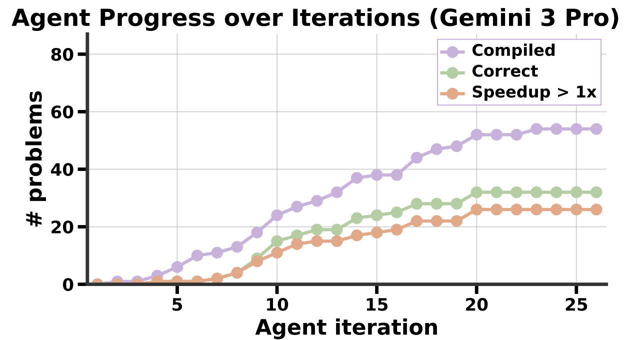


Figure 7. **Test-time compute enables hill-climbing on PKB.** We provide Gemini 3 Pro with a Mini-SWE-Agent harness and terminal environment, allowing it to make arbitrary changes to a codebase. Achieves 7 more correct (31) and 14 more $fast_1$ (26) solutions than single-shot.

lack fluency with CUDA and Torch APIs. Reasoning models largely overcome this, with compilation rates exceeding 50%, yet correctness remains low; the majority of kernels produce incorrect outputs, suggesting models cannot reliably reason about rank coordination, data partitioning, and collective ordering. The shift from syntax failures to correctness failures across model families reveals that the core bottleneck is not code generation fluency but a deeper failure to reason about parallel execution.

5.2. Speedup Distribution

As shown in Figure 6, fewer than 30% of one-shot LLM-generated kernels are both correct and faster than the PyTorch + NCCL reference. Figure 4 highlights how, once correctness is established, speedups naturally arise from eliminating NCCL staging overhead in favor of direct NVLink loads and stores. Analyzing the SASS assembly of correctly compiled kernels (Figure 5), we find that the vast majority rely on copy engine and SM load/store instructions for inter-GPU transfers, while more advanced mechanisms — TMA, `multimem`, and intra-SM warp specialization — are nearly absent. This suggests that one-shot models gravitate toward simple transfer primitives and have not learned to compose fine-grained overlap techniques that unlock peak hardware performance.

5.3. Impact of Agentic Harness & Iterative Feedback

Having established that base models struggle significantly on ParallelKernelBench, we now investigate whether iterative feedback from an agentic harness can improve performance.

Setup. For the agentic setting, we wrap our generation pipeline in a multi-turn harness built on mini-swe-agent (Yang et al., 2024), giving one of the best-performing base models (Gemini 3 Pro) access to a local bash environment rooted at the repository. Building on the one-shot prompt, the agent can compile, evaluate for correctness and speedup via a provided script, and iteratively refine its kernel via a local terminal environment. For more details on the harness, see Appendix F.

Performance. Figure 7 shows that iterative feedback allows the agent to hill-climb beyond single-shot performance, though gains quickly plateau—revealing a ceiling imposed by the model's capacity to reason about distributed kernel optimization. The agent solved 35 of 87 problems (40.22%), up from 24 in the single-shot baseline, with 26 achieving speedup $> 1\times$ over the reference.

6. Qualitative Kernel Analysis

In this section, we highlight common themes among LLM-generated kernels that achieved notable speedups over the PyTorch baseline, including several net-new kernels with no existing optimized open-source implementation, and discuss the broader implications. Detailed kernel analyses are provided in Appendix G.

NVLink P2P Data Accesses. The most consistent source of speedup across problems comes from eliminating collective staging and packing overhead entirely in favor of direct peer memory access via SM load/store instructions, transparently routed through NVSwitch. Rather than paying for NCCL's staging overhead, kernels read remote ranks' symmetric memory buffers as pointer dereferences, fusing communication directly into the kernel. An example of this behavior is provided in Figure 8.

Movement Efficiency. Avoiding slow networked transfers—or batching them into as few wide chunks as possible—reduces execution time. LLMs generated kernels for several collective operations that matched or surpassed NCCL at small (1024×1024) tensor sizes (e.g., AllGather at $1.2\times$, Scatter at $1.6\times$) by dynamically selecting the widest available memory instructions based on alignment, with certain MoE and Ulysses kernels further exploiting a "fast path" of 128-bit aligned vectorized loads across NVLink.

Pipelining. Once NVLink bandwidth is the bottleneck, the goal is to hide latency by overlapping computation with communication. In the generated kernels, we found examples

PyTorch Reference:

```python
def ulysses_allgather_reference():
  x = x.contiguous()
  x_size = torch.tensor(x.size()...)
  # [1] Gather tensor sizes
  size_list = [torch.zeros(...) for ...]
  dist.all_gather(size_list, x_size)
  # [2] Gather tensor data
  tensor_list = [torch.empty(...) ...]
  dist.all_gather(
    tensor_list, x, group=group)
```

LLM-Generated Solution:

```cuda
// Replaces [1] + [2]: single kernel
// with direct NVLink loads
template<typename T>
__global__ void
ulysses_allgather_kernel(...):
  int64_t src_idx = ...
  int64_t dst_idx = ...
  // One-sided peer memory read
  out[dst_idx] = src[src_idx];

ulysses_allgather_kernel
  <<<blocks, threads, 0>>>(...)
```

Figure 8. **LLM eliminates NCCL staging overhead in Ulysses all-gather via direct NVLink loads.** The reference implementation uses two `dist.all_gather` calls routed through NCCL's staging buffers: one to exchange tensor sizes and one for the data itself. The LLM-generated solution replaces both with a single CUDA kernel that reads directly from peer GPU memory over NVLink, bypassing NCCL entirely and eliminating the intermediate copy.

(e.g. Ulysses All-to-All ($1.84\times$)) of the LLM attempting this using separate CUDA streams: each data chunk is processed on its own CUDA stream, allowing compute on one chunk to overlap with NVLink transfers for the next.

7. Conclusion

We present three contributions: (1) an organization of distributed AI workloads by parallelism strategy; (2) ParallelKernelBench, a benchmark of 87 multi-GPU problems evaluated against compute, memory, and interconnect ceilings; (3) an analysis of frontier LLM-generated kernels, revealing where models succeed and fall short in communication-aware optimization. Crucially, all baselines use standard PyTorch and NCCL collectives, ensuring that they remain valid as new hardware topologies emerge. For these reasons, PKB provides a concrete target toward the longer-term goal of LLM systems that can autonomously optimize and manage large-scale distributed infrastructure. We provide limitations and directions for future work in Appendix H.

References

Nemo rl: A scalable and efficient post-training library. https://github.com/NVIDIA-NeMo/RL, 2025. GitHub repository.

ai, S., Teng, H., Jia, H., Sun, L., Li, L., Li, M., Tang, M., Han, S., Zhang, T., Zhang, W. Q., Luo, W., Kang, X., Sun, Y., Cao, Y., Huang, Y., Lin, Y., Fang, Y., Tao, Z., Zhang, Z., Wang, Z., Liu, Z., Shi, D., Su, G., Sun, H., Pan, H., Wang, J., Sheng, J., Cui, M., Hu, M., Yan, M., Yin, S., Zhang, S., Liu, T., Yin, X., Yang, X., Song, X., Hu, X., Zhang, Y., and Li, Y. Magi-1: Autoregressive video generation at scale, 2025. URL https://arxiv.org/abs/2505.13211.

Aimuyo, O. J., Oh, B., and Singh, R. Flashmoe: Fast distributed moe in a single kernel, 2025. URL https://arxiv.org/abs/2506.04667.

Anthony, Q., Michalowicz, B., Hatef, J., Xu, L., Abduljabbar, M., Shafi, A., Subramoni, H., and Panda, D. Demystifying the communication characteristics for distributed transformer models, 2024. URL https://arxiv.org/abs/2408.10197.

Awad, M., Osama, M., and Potter, B. Iris: First-class multi-gpu programming experience in triton, 2025. URL https://arxiv.org/abs/2511.12500.

Baronio, C., Marsella, P., Pan, B., Guo, S., and Alberti, S. Kevin: Multi-turn rl for generating cuda kernels, 2025. URL https://arxiv.org/abs/2507.11948.

Bhargava, R., Allison, T., and Petty, H. Nvidia vera rubin pod: Seven chips, five rack-scale systems, one ai supercomputer, Apr 2026. URL https://developer.nvidia.com/blog/nvidia-vera-rubin-pod-seven-chips-five-rack-scale-systems-one-ai-supercomputer/.

Bonev, B., Kurth, T., Hundt, C., Pathak, J., Baust, M., Kashinath, K., and Anandkumar, A. Spherical fourier neural operators: Learning stable dynamics on the sphere, 2023. URL https://arxiv.org/abs/2306.03838.

Cardwell, D. and Song, F. An extended roofline model with communication-awareness for distributed-memory hpc systems. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, HPCAsia '19, pp. 26–35, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450366328. doi: 10.1145/3293320.3293321. URL https://doi.org/10.1145/3293320.3293321.

Carion, N., Gustafson, L., Hu, Y.-T., Debnath, S., Hu, R., Suris, D., Ryali, C., Alwala, K. V., Khedr, H., Huang, A., Lei, J., Ma, T., Guo, B., Kalla, A., Marks, M., Greer, J., Wang, M., Sun, P., Rädle, R., Afouras, T., Mavroudi, E., Xu, K., Wu, T.-H., Zhou, Y., Momeni, L., Hazra, R., Ding, S., Vaze, S., Porcher, F., Li, F., Li, S., Kamath, A., Cheng, H. K., Dollár, P., Ravi, N., Saenko, K., Zhang, P., and Feichtenhofer, C. Sam 3: Segment anything with concepts, 2025. URL https://arxiv.org/abs/2511.16719.

Checconi, F., Tithi, J. J., and Petrini, F. Ridgeline: A 2d roofline model for distributed systems, 2025. URL https://arxiv.org/abs/2209.01368.

Czaja, J., Gallus, M., Wozna, J., Grygielski, A., and Tao, L. Applying the roofline model for deep learning performance optimizations, 2020. URL https://arxiv.org/abs/2009.11224.

Dai, W., Wu, H., Yu, Q., ang Gao, H., Li, J., Jiang, C., Lou, W., Song, Y., Yu, H., Chen, J., Ma, W.-Y., Zhang, Y.-Q., Liu, J., Wang, M., Liu, X., and Zhou, H. Cuda agent: Large-scale agentic rl for high-performance cuda kernel generation, 2026. URL https://arxiv.org/abs/2602.24286.

DeepSeek-AI, Liu, A., Feng, B., Xue, B., Wang, B., Wu, B., Lu, C., Zhao, C., Deng, C., Zhang, C., Ruan, C., Dai, D., Guo, D., Yang, D., Chen, D., Ji, D., Li, E., Lin, F., Dai, F., Luo, F., Hao, G., Chen, G., Li, G., Zhang, H., Bao, H., Xu, H., Wang, H., Zhang, H., Ding, H., Xin, H., Gao, H., Li, H., Qu, H., Cai, J. L., Liang, J., Guo, J., Ni, J., Li, J., Wang, J., Chen, J., Chen, J., Yuan, J., Qiu, J., Li, J., Song, J., Dong, K., Hu, K., Gao, K., Guan, K., Huang, K., Yu, K., Wang, L., Zhang, L., Xu, L., Xia, L., Zhao, L., Wang, L., Zhang, L., Li, M., Wang, M., Zhang, M., Zhang, M., Tang, M., Li, M., Tian, N., Huang, P., Wang, P., Zhang, P., Wang, Q., Zhu, Q., Chen, Q., Du, Q., Chen, R. J., Jin, R. L., Ge, R., Zhang, R., Pan, R., Wang, R., Xu, R., Zhang, R., Chen, R., Li, S. S., Lu, S., Zhou, S., Chen, S., Wu, S., Ye, S., Ye, S., Ma, S., Wang, S., Zhou, S., Yu, S., Zhou, S., Pan, S., Wang, T., Yun, T., Pei, T., Sun, T., Xiao, W. L., Zeng, W., Zhao, W., An, W., Liu, W., Liang, W., Gao, W., Yu, W., Zhang, W., Li, X. Q., Jin, X., Wang, X., Bi, X., Liu, X., Wang, X., Shen, X., Chen, X., Zhang, X., Chen, X., Nie, X., Sun, X., Wang, X., Cheng, X., Liu, X., Xie, X., Liu, X., Yu, X., Song, X., Shan, X., Zhou, X., Yang, X., Li, X., Su, X., Lin, X., Li, Y. K., Wang, Y. Q., Wei, Y. X., Zhu, Y. X., Zhang, Y., Xu, Y., Xu, Y., Huang, Y., Li, Y., Zhao, Y., Sun, Y., Li, Y., Wang, Y., Yu, Y., Zheng, Y., Zhang, Y., Shi, Y., Xiong, Y., He, Y., Tang, Y., Piao, Y., Wang, Y., Tan, Y., Ma, Y., Liu, Y., Guo, Y., Wu, Y., Ou, Y., Zhu, Y., Wang, Y., Gong, Y., Zou, Y., He, Y., Zha, Y., Xiong, Y., Ma, Y., Yan, Y., Luo, Y., You, Y., Liu, Y., Zhou, Y., Wu, Z. F., Ren, Z. Z., Ren, Z., Sha, Z., Fu, Z., Xu, Z., Huang, Z., Zhang, Z., Xie, Z., Zhang, Z., Hao, Z., Gou, Z., Ma, Z.,

Yan, Z., Shao, Z., Xu, Z., Wu, Z., Zhang, Z., Li, Z., Gu, Z., Zhu, Z., Liu, Z., Li, Z., Xie, Z., Song, Z., Gao, Z., and Pan, Z. Deepseek-v3 technical report, 2025. URL https://arxiv.org/abs/2412.19437.

Duan, J., Zhang, S., Wang, Z., Jiang, L., Qu, W., Hu, Q., Wang, G., Weng, Q., Yan, H., Zhang, X., Qiu, X., Lin, D., Wen, Y., Jin, X., Zhang, T., and Sun, P. Efficient training of large language models on distributed infrastructures: A survey, 2024. URL https://arxiv.org/abs/2407.20018.

Fernandez, J., Wehrstedt, L., Shamis, L., Elhoushi, M., Saladi, K., Bisk, Y., Strubell, E., and Kahn, J. Hardware scaling trends and diminishing returns in large-scale distributed training, 2025. URL https://arxiv.org/abs/2411.13055.

Fey, M. and Lenssen, J. E. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

GLM-5-Team, :, Zeng, A., Lv, X., Hou, Z., Du, Z., Zheng, Q., Chen, B., Yin, D., Ge, C., Huang, C., Xie, C., Zhu, C., Yin, C., Wang, C., Pan, G., Zeng, H., Zhang, H., Wang, H., Chen, H., Zhang, J., Jiao, J., Guo, J., Wang, J., Du, J., Wu, J., Wang, K., Li, L., Fan, L., Zhong, L., Liu, M., Zhao, M., Du, P., Dong, Q., Lu, R., Shuang-Li, Cao, S., Liu, S., Jiang, T., Chen, X., Zhang, X., Huang, X., Dong, X., Xu, Y., Wei, Y., An, Y., Niu, Y., Zhu, Y., Wen, Y., Cen, Y., Bai, Y., Qiao, Z., Wang, Z., Wang, Z., Zhu, Z., Liu, Z., Li, Z., Wang, B., Wen, B., Huang, C., Cai, C., Yu, C., Li, C., Hu, C., Zhang, C., Zhang, D., Lin, D., Yang, D., Wang, D., Ai, D., Zhu, E., Yi, F., Chen, F., Wen, G., Sun, H., Zhao, H., Hu, H., Zhang, H., Liu, H., Zhang, H., Peng, H., Tai, H., Zhang, H., Liu, H., Wang, H., Yan, H., Ge, H., Liu, H., Chu, H., Zhao, J., Wang, J., Zhao, J., Ren, J., Wang, J., Zhang, J., Gui, J., Zhao, J., Li, J., An, J., Li, J., Yuan, J., Du, J., Liu, J., Zhi, J., Duan, J., Zhou, K., Wei, K., Wang, K., Luo, K., Zhang, L., Sha, L., Xu, L., Wu, L., Ding, L., Chen, L., Li, M., Lin, N., Ta, P., Zou, Q., Song, R., Yang, R., Tu, S., Yang, S., Wu, S., Zhang, S., Li, S., Li, S., Fan, S., Qin, W., Tian, W., Zhang, W., Yu, W., Liang, W., Kuang, X., Cheng, X., Li, X., Yan, X., Hu, X., Ling, X., Fan, X., Xia, X., Zhang, X., Zhang, X., Pan, X., Zou, X., Zhang, X., Liu, Y., Wu, Y., Li, Y., Wang, Y., Zhu, Y., Tan, Y., Zhou, Y., Pan, Y., Zhang, Y., Su, Y., Geng, Y., Yan, Y., Tan, Y., Bi, Y., Shen, Y., Yang, Y., Li, Y., Liu, Y., Wang, Y., Li, Y., Wu, Y., Zhang, Y., Duan, Y., Zhang, Y., Liu, Z., Jiang, Z., Yan, Z., Zhang, Z., Wei, Z., Chen, Z., Feng, Z., Yao, Z., Chai, Z., Wang, Z., Zhang, Z., Xu, B., Huang, M., Wang, H., Li, J., Dong, Y., and Tang, J. Glm-5: from vibe coding to agentic engineering, 2026. URL https://arxiv.org/abs/2602.15763.

Gond, R., Kwatra, N., and Ramjee, R. Tokenweave: Efficient compute-communication overlap for distributed llm inference, 2025. URL https://arxiv.org/abs/2505.11329.

Grattafiori, A., Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Vaughan, A., Yang, A., Fan, A., Goyal, A., Hartshorn, A., Yang, A., Mitra, A., Sravankumar, A., Korenev, A., Hinsvark, A., Rao, A., Zhang, A., Rodriguez, A., Gregerson, A., Spataru, A., Roziere, B., Biron, B., Tang, B., Chern, B., Caucheteux, C., Nayak, C., Bi, C., Marra, C., McConnell, C., Keller, C., Touret, C., Wu, C., Wong, C., Ferrer, C. C., Nikolaidis, C., Allonsius, D., Song, D., Pintz, D., Livshits, D., Wyatt, D., Esiobu, D., Choudhary, D., Mahajan, D., Garcia-Olano, D., Perino, D., Hupkes, D., Lakomkin, E., AlBadawy, E., Lobanova, E., Dinan, E., Smith, E. M., Radenovic, F., Guzmán, F., Zhang, F., Synnaeve, G., Lee, G., Anderson, G. L., Thattai, G., Nail, G., Mialon, G., Pang, G., Cucurell, G., Nguyen, H., Korevaar, H., Xu, H., Touvron, H., Zarov, I., Ibarra, I. A., Kloumann, I., Misra, I., Evtimov, I., Zhang, J., Copet, J., Lee, J., Geffert, J., Vranes, J., Park, J., Mahadeokar, J., Shah, J., van der Linde, J., Billock, J., Hong, J., Lee, J., Fu, J., Chi, J., Huang, J., Liu, J., Wang, J., Yu, J., Bitton, J., Spisak, J., Park, J., Rocca, J., Johnstun, J., Saxe, J., Jia, J., Alwala, K. V., Prasad, K., Upasani, K., Plawiak, K., Li, K., Heafield, K., Stone, K., El-Arini, K., Iyer, K., Malik, K., Chiu, K., Bhalla, K., Lakhotia, K., Rantala-Yeary, L., van der Maaten, L., Chen, L., Tan, L., Jenkins, L., Martin, L., Madaan, L., Malo, L., Blecher, L., Landzaat, L., de Oliveira, L., Muzzi, M., Pasupuleti, M., Singh, M., Paluri, M., Kardas, M., Tsimpoukelli, M., Oldham, M., Rita, M., Pavlova, M., Kambadur, M., Lewis, M., Si, M., Singh, M. K., Hassan, M., Goyal, N., Torabi, N., Bashlykov, N., Bogoychev, N., Chatterji, N., Zhang, N., Duchenne, O., Çelebi, O., Alrassy, P., Zhang, P., Li, P., Vasic, P., Weng, P., Bhargava, P., Dubal, P., Krishnan, P., Koura, P. S., Xu, P., He, Q., Dong, Q., Srinivasan, R., Ganapathy, R., Calderer, R., Cabral, R. S., Stojnic, R., Raileanu, R., Maheswari, R., Girdhar, R., Patel, R., Sauvestre, R., Polidoro, R., Sumbaly, R., Taylor, R., Silva, R., Hou, R., Wang, R., Hosseini, S., Chennabasappa, S., Singh, S., Bell, S., Kim, S. S., Edunov, S., Nie, S., Narang, S., Raparthy, S., Shen, S., Wan, S., Bhosale, S., Zhang, S., Vandenhende, S., Batra, S., Whitman, S., Sootla, S., Collot, S., Gururangan, S., Borodinsky, S., Herman, T., Fowler, T., Sheasha, T., Georgiou, T., Scialom, T., Speckbacher, T., Mihaylov, T., Xiao, T., Karn, U., Goswami, V., Gupta, V., Ramanathan, V., Kerkez, V., Gonguet, V., Do, V., Vogeti, V., Albiero, V., Petrovic, V., Chu, W., Xiong, W., Fu, W., Meers, W., Martinet, X., Wang, X., Wang, X., Tan, X. E., Xia, X., Xie, X., Jia, X., Wang, X., Goldschlag, Y., Gaur, Y., Babaei, Y., Wen, Y., Song, Y., Zhang,

550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604

Y., Li, Y., Mao, Y., Coudert, Z. D., Yan, Z., Chen, Z., Papakipos, Z., Singh, A., Srivastava, A., Jain, A., Kelsey, A., Shajnfeld, A., Gangidi, A., Victoria, A., Goldstand, A., Menon, A., Sharma, A., Boesenberg, A., Baevski, A., Feinstein, A., Kallet, A., Sangani, A., Teo, A., Yunus, A., Lupu, A., Alvarado, A., Caples, A., Gu, A., Ho, A., Poulton, A., Ryan, A., Ramchandani, A., Dong, A., Franco, A., Goyal, A., Saraf, A., Chowdhury, A., Gabriel, A., Bharambe, A., Eisenman, A., Yazdan, A., James, B., Maurer, B., Leonhardi, B., Huang, B., Loyd, B., Paola, B. D., Paranjape, B., Liu, B., Wu, B., Ni, B., Hancock, B., Wasti, B., Spence, B., Stojkovic, B., Gamido, B., Montalvo, B., Parker, C., Burton, C., Mejia, C., Liu, C., Wang, C., Kim, C., Zhou, C., Hu, C., Chu, C.-H., Cai, C., Tindal, C., Feichtenhofer, C., Gao, C., Civin, D., Beaty, D., Kreymer, D., Li, D., Adkins, D., Xu, D., Testuggine, D., David, D., Parikh, D., Liskovich, D., Foss, D., Wang, D., Le, D., Holland, D., Dowling, E., Jamil, E., Montgomery, E., Presani, E., Hahn, E., Wood, E., Le, E.-T., Brinkman, E., Arcaute, E., Dunbar, E., Smothers, E., Sun, F., Kreuk, F., Tian, F., Kokkinos, F., Ozgenel, F., Caggioni, F., Kanayet, F., Seide, F., Florez, G. M., Schwarz, G., Badeer, G., Swee, G., Halpern, G., Herman, G., Sizov, G., Guangyi, Zhang, Lakshminarayanan, G., Inan, H., Shojanazeri, H., Zou, H., Wang, H., Zha, H., Habeeb, H., Rudolph, H., Suk, H., Aspegren, H., Goldman, H., Zhan, H., Damlaj, I., Molybog, I., Tufanov, I., Leontiadis, I., Veliche, I.-E., Gat, I., Weissman, J., Geboski, J., Kohli, J., Lam, J., Asher, J., Gaya, J.-B., Marcus, J., Tang, J., Chan, J., Zhen, J., Reizenstein, J., Teboul, J., Zhong, J., Jin, J., Yang, J., Cummings, J., Carvill, J., Shepard, J., McPhie, J., Torres, J., Ginsburg, J., Wang, J., Wu, K., U, K. H., Saxena, K., Khandelwal, K., Zand, K., Matosich, K., Veeraraghavan, K., Michelena, K., Li, K., Jagadeesh, K., Huang, K., Chawla, K., Huang, K., Chen, L., Garg, L., A, L., Silva, L., Bell, L., Zhang, L., Guo, L., Yu, L., Moshkovich, L., Wehrstedt, L., Khabsa, M., Avalani, M., Bhatt, M., Mankus, M., Hasson, M., Lennie, M., Reso, M., Groshev, M., Naumov, M., Lathi, M., Keneally, M., Liu, M., Seltzer, M. L., Valko, M., Restrepo, M., Patel, M., Vyatskov, M., Samvelyan, M., Clark, M., Macey, M., Wang, M., Hermoso, M. J., Metanat, M., Rastegari, M., Bansal, M., Santhanam, N., Parks, N., White, N., Bawa, N., Singhal, N., Egebo, N., Usunier, N., Mehta, N., Laptev, N. P., Dong, N., Cheng, N., Chernoguz, O., Hart, O., Salpekar, O., Kalinli, O., Kent, P., Parekh, P., Saab, P., Balaji, P., Rittner, P., Bontrager, P., Roux, P., Dollar, P., Zvyagina, P., Ratanchandani, P., Yuvraj, P., Liang, Q., Alao, R., Rodriguez, R., Ayub, R., Murthy, R., Nayani, R., Mitra, R., Parthasarathy, R., Li, R., Hogan, R., Battey, R., Wang, R., Howes, R., Rinott, R., Mehta, S., Siby, S., Bondu, S. J., Datta, S., Chugh, S., Hunt, S., Dhillon, S., Sidorov, S., Pan, S., Mahajan, S., Verma, S., Yamamoto, S., Ramaswamy, S., Lindsay, S., Lindsay, S., Feng, S., Lin, S., Zha, S. C., Patil, S., Shankar, S., Zhang, S., Zhang, S., Wang, S., Agarwal, S., Sajuyigbe, S., Chintala, S., Max, S., Chen, S., Kehoe, S., Satterfield, S., Govindaprasad, S., Gupta, S., Deng, S., Cho, S., Virk, S., Subramanian, S., Choudhury, S., Goldman, S., Remez, T., Glaser, T., Best, T., Koehler, T., Robinson, T., Li, T., Zhang, T., Matthews, T., Chou, T., Shaked, T., Vontimitta, V., Ajayi, V., Montanez, V., Mohan, V., Kumar, V. S., Mangla, V., Ionescu, V., Poenaru, V., Mihailescu, V. T., Ivanov, V., Li, W., Wang, W., Jiang, W., Bouaziz, W., Constable, W., Tang, X., Wu, X., Wang, X., Wu, X., Gao, X., Kleinman, Y., Chen, Y., Hu, Y., Jia, Y., Qi, Y., Li, Y., Zhang, Y., Zhang, Y., Adi, Y., Nam, Y., Yu Wang, Zhao, Y., Hao, Y., Qian, Y., Li, Y., He, Y., Rait, Z., DeVito, Z., Rosnbrick, Z., Wen, Z., Yang, Z., Zhao, Z., and Ma, Z. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

Hu, B., V, Y. S. V., Agarwal, S., and Akella, A. Cuco: An agentic framework for compute and communication co-design, 2026a. URL https://arxiv.org/abs/2603.02376.

Hu, Z., Shen, S., Bonato, T., Jeaugey, S., Alexander, C., Spada, E., Dinan, J., Hammond, J., and Hoefler, T. Demystifying nccl: An in-depth analysis of gpu communication protocols and algorithms, 2026b. URL https://arxiv.org/abs/2507.04786.

Ilharco, G., Wortsman, M., Wightman, R., Gordon, C., Carlini, N., Taori, R., Dave, A., Shankar, V., Namkoong, H., Miller, J., Hajishirzi, H., Farhadi, A., and Schmidt, L. Openclip, July 2021. URL https://doi.org/10.5281/zenodo.5143773. If you use this software, please cite it as below.

Ivchenko, D., Van Der Staay, D., Taylor, C., Liu, X., Feng, W., Kindi, R., Sudarshan, A., and Sefati, S. Torchrec: a pytorch domain library for recommendation systems. In *Proceedings of the 16th ACM Conference on Recommender Systems*, RecSys '22, pp. 482–483, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392785. doi: 10.1145/3523227.3547387. URL https://doi.org/10.1145/3523227.3547387.

Jacobs, S. A., Tanaka, M., Zhang, C., Zhang, M., Song, S. L., Rajbhandari, S., and He, Y. Deepspeed ulysses: System optimizations for enabling training of extreme long sequence transformer models, 2023. URL https://arxiv.org/abs/2309.14509.

Jiang, Z., Lin, H., Zhong, Y., Huang, Q., Chen, Y., Zhang, Z., Peng, Y., Li, X., Xie, C., Nong, S., Jia, Y., He, S., Chen, H., Bai, Z., Hou, Q., Yan, S., Zhou, D., Sheng, Y., Jiang, Z., Xu, H., Wei, H., Zhang, Z., Nie, P., Zou,

L., Zhao, S., Xiang, L., Liu, Z., Li, Z., Jia, X., Ye, J., Jin, X., and Liu, X. Megascale: Scaling large language model training to more than 10,000 gpus, 2024. URL https://arxiv.org/abs/2402.15627.

Jouppi, N. P., Kurian, G., Li, S., Ma, P., Nagarajan, R., Nai, L., Patil, N., Subramanian, S., Swing, A., Towles, B., Young, C., Zhou, X., Zhou, Z., and Patterson, D. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings, 2023. URL https://arxiv.org/abs/2304.01433.

Kernel, S. Reimagining kernel generation at the ptx layer: An llm system learning from dsls to outperform them. Standard Kernel Blog, April 2026. URL https://standardkernel.com/blog/reimagining-kernel-generation-at-the-ptx-layer-learning-from-and-outperforming-dsls/. Accessed: 2026-05-02.

Krashinsky, R., Giroux, O., Jones, S., Stam, N., and Ramaswamy, S. Nvidia ampere architecture in-depth, May 2023. URL https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/.

Lange, R. T., Sun, Q., Prasad, A., Faldor, M., Tang, Y., and Ha, D. Towards robust agentic cuda kernel benchmarking, verification, and optimization, 2025. URL https://arxiv.org/abs/2509.14279.

Li, M., Cai, T., Cao, J., Zhang, Q., Cai, H., Bai, J., Jia, Y., Liu, M.-Y., Li, K., and Han, S. Distrifusion: Distributed parallel inference for high-resolution diffusion models, 2024. URL https://arxiv.org/abs/2402.19481.

Lin, E., Modi, S., Hari, S. K. S., Huang, Q., Ye, Z., Qin, N., Zhou, F., Zhang, Y., Wang, J., Damani, S., Peri, D., Xie, O., Kane, A., Maor, M., Behar, M., Cao, T., Mehta, R., Singh, V., Mailthody, V. S., Chen, T., Ye, Z., Chen, H., Chen, T., Grover, V., Chen, W., Liu, W., Chung, E., Ceze, L., Bringmann, R., Zeller, C., Lightstone, M., Kozyrakis, C., and Shi, H. Sol-execbench: Speed-of-light benchmarking for real-world gpu kernels against hardware limits, 2026. URL https://arxiv.org/abs/2603.19173.

Lin, J., Feng, X., Qi, H., Liu, Y., Ling, C., Tang, B., Wang, Y., Tao, X., and Li, W. Flux: Fine-grained communication scheduling for distributed training in multi-tenant ai clusters. In *2025 IEEE/ACM 33rd International Symposium on Quality of Service (IWQoS)*, pp. 1–2, 2025. doi: 10.1109/IWQoS65803.2025.11143475.

Liu, H., Zaharia, M., and Abbeel, P. Ring attention with blockwise transformers for near-infinite context, 2023. URL https://arxiv.org/abs/2310.01889.

Liu, J., Su, J., Yao, X., Jiang, Z., Lai, G., Du, Y., Qin, Y., Xu, W., Lu, E., Yan, J., Chen, Y., Zheng, H., Liu, Y., Liu, S., Yin, B., He, W., Zhu, H., Wang, Y., Wang, J., Dong, M., Zhang, Z., Kang, Y., Zhang, H., Xu, X., Zhang, Y., Wu, Y., Zhou, X., and Yang, Z. Muon is scalable for llm training, 2025. URL https://arxiv.org/abs/2502.16982.

Mitrasish. Nvidia rubin r100: Specs, architecture, and gpu cloud availability: Spheron blog, Apr 2026. URL https://www.spheron.network/blog/nvidia-rubin-r100-guide/.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V. A., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., Phanishayee, A., and Zaharia, M. Efficient large-scale language model training on gpu clusters using megatron-lm, 2021. URL https://arxiv.org/abs/2104.04473.

NVIDIA. cupynumeric. https://github.com/nv-legate/cupynumeric, a. GitHub repository, accessed 2026-05-06.

NVIDIA. Nvidia tensorrt-llm. https://github.com/NVIDIA/TensorRT-LLM, b. GitHub repository, accessed 2026-05-06.

Oquab, M., Darcet, T., Moutakanni, T., Vo, H., Szafraniec, M., Khalidov, V., Fernandez, P., Haziza, D., Massa, F., El-Nouby, A., Assran, M., Ballas, N., Galuba, W., Howes, R., Huang, P.-Y., Li, S.-W., Misra, I., Rabbat, M., Sharma, V., Synnaeve, G., Xu, H., Jegou, H., Mairal, J., Labatut, P., Joulin, A., and Bojanowski, P. Dinov2: Learning robust visual features without supervision, 2024. URL https://arxiv.org/abs/2304.07193.

Ouyang, A., Guo, S., Arora, S., Zhang, A. L., Hu, W., Ré, C., and Mirhoseini, A. Kernelbench: Can llms write efficient gpu kernels?, 2025. URL https://arxiv.org/abs/2502.10517.

PhysicsNeMo Contributors. Nvidia physicsnemo: An open-source framework for physics-based deep learning in science and engineering. https://github.com/NVIDIA/physicsnemo, 2023. GitHub repository.

Poli, M., Massaroli, S., Nguyen, E., Fu, D. Y., Dao, T., Baccus, S., Bengio, Y., Ermon, S., and Ré, C. Hyena hierarchy: Towards larger convolutional language models. In *International Conference on Machine Learning*, pp. 28043–28078. PMLR, 2023.

PyTorch Team. Pytorch symmetric memory. https://docs.pytorch.org/docs/stable/symmetric_memory.html, 2026. Accessed: 2026-04-27.

Schieffer, G., Wahlgren, J., Shi, R., Leon, E. A., Pearce, R., Gokhale, M., and Peng, I. Inter-apu communication on amd mi300a systems via infinity fabric: A deep dive. In *Proceedings of the International Symposium on Memory Systems*, MemSys '25, pp. 238–250, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400720024. doi: 10.1145/3767110.3767130. URL https://doi.org/10.1145/3767110.3767130.

Stanford CS336. Cs336 spring 2026 assignment 2: Systems. https://github.com/stanford-cs336/assignment2-systems, 2026. GitHub repository.

Sul, S. and Re, C. Thunderkittens 2.0: Even faster kernels for your gpus, Feb 2026. URL https://hazyresearch.stanford.edu/blog/2026-02-19-tk-2.

Sul, S. H., Arora, S., Spector, B. F., and Ré, C. Parallelkittens: Systematic and practical simplification of multi-gpu ai kernels, 2025. URL https://arxiv.org/abs/2511.13940.

Wang, H., Zhang, L., Han, J., and E, W. Deepmd-kit: A deep learning package for many-body potential energy representation and molecular dynamics. *Computer Physics Communications*, 228:178–184, 2018. ISSN 0010-4655. doi: 10.1016/j.cpc.2018.03.016. URL http://dx.doi.org/10.1016/j.cpc.2018.03.016.

Wang, M., Zheng, D., Ye, Z., Gan, Q., Li, M., Song, X., Zhou, J., Ma, C., Yu, L., Gai, Y., Xiao, T., He, T., Karypis, G., Li, J., and Zhang, Z. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019.

Wen, Z., Zhang, Y., Li, Z., Liu, Z., Xie, L., and Zhang, T. Multikernelbench: A multi-platform benchmark for kernel generation, 2025. URL https://arxiv.org/abs/2507.17773.

Williams, S., Waterman, A., and Patterson, D. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009. ISSN 0001-0782. doi: 10.1145/1498765.1498785. URL https://doi.org/10.1145/1498765.1498785.

Xin, J., Canini, M., Richtárik, P., and Horváth, S. Global-qsgd: Allreduce-compatible quantization for distributed learning with theoretical guarantees. In *Proceedings of the 5th Workshop on Machine Learning and Systems*, EuroMLSys '25, pp. 216–229, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400715389. doi: 10.1145/3721146.3721932. URL https://doi.org/10.1145/3721146.3721932.

Xing, S., Zhai, Y., Jiang, A., Dong, Y., Wu, Y., Ye, Z., Ruan, C., Huang, Y., Zhang, Y., Yin, L., Bayyapu, A., Ceze, L., and Chen, T. Flashinfer-bench: Building the virtuous cycle for ai-driven llm systems, 2026. URL https://arxiv.org/abs/2601.00227.

Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K. R., and Press, O. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2405.15793.

Yang, S. and Zhang, Y. Fla: A triton-based library for hardware-efficient implementations of linear attention mechanism, January 2024. URL https://github.com/fla-org/flash-linear-attention.

Ye, V., Li, R., Kerr, J., Turkulainen, M., Yi, B., Pan, Z., Seiskari, O., Ye, J., Hu, J., Tancik, M., and Kanazawa, A. gsplat: An open-source library for gaussian splatting, 2024. URL https://arxiv.org/abs/2409.06765.

yifuwang, He, H., and Wehrstedt, L. Pytorch symmetricmemory: Harnessing nvlink programmability with ease, Feb 2025. URL https://dev-discuss.pytorch.org/t/pytorch-symmetricmemory-harnessing-nvlink-program 2798.

Zhao, C., Zhou, S., Zhang, L., Deng, C., Xu, Z., Liu, Y., Yu, K., Li, J., and Zhao, L. Deepep: an efficient expert-parallel communication library. https://github.com/deepseek-ai/DeepEP, 2025.

Zheng, D., Song, X., Zhu, Q., Zhang, J., Vasiloudis, T., Ma, R., Zhang, H., Wang, Z., Adeshina, S., Nisa, I., Mottini, A., Cui, Q., Rangwala, H., Zeng, B., Faloutsos, C., and Karypis, G. Graphstorm: All-in-one graph machine learning framework for industry applications. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '24, pp. 6356–6367, 2024a. doi: 10.1145/3637528.3671603. URL https://doi.org/10.1145/3637528.3671603.

Zheng, S., Bao, W., Hou, Q., Zheng, X., Fang, J., Huang, C., Li, T., Duanmu, H., Chen, R., Xu, R., Guo, Y., Zheng, N., Jiang, Z., Di, X., Wang, D., Ye, J., Lin, H., Chang, L.-W., Lu, L., Liang, Y., Zhai, J., and Liu, X. Triton-distributed: Programming overlapping kernels on distributed ai systems with the triton compiler, 2025. URL https://arxiv.org/abs/2504.19442.

Zheng, Z., Peng, X., Yang, T., Shen, C., Li, S., Liu, H., Zhou, Y., Li, T., and You, Y. Open-sora: Democratizing efficient video production for all, 2024b. URL https://arxiv.org/abs/2412.20404.

A. Prompt & In-Context Example

To prompt frontier models, we provide a single in-context example: a symmetric-memory element-wise add. This was chosen to be the simplest possible kernel that demonstrates the two primitives PKB problems require: the symmetric heap API for allocating peer-addressable buffers, and direct NVLink P2P data transfer via device pointers. All other problem-solving is left to the model.

```python
import torch
import torch.distributed as dist
import torch.distributed._symmetric_memory as symm_mem
from utils.cuda_helpers import compile_cuda_extension

CUDA_SRC = r'''
#include <torch/extension.h>
#include <ATen/cuda/CUDAContext.h>
#include <cuda_runtime.h>

__global__ void symmetric_add_kernel(
    const float* __restrict__ local_data,
    const float* __restrict__ remote_data,
    float* __restrict__ out,
    int64_t n
) {
    int64_t idx = (int64_t)blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        out[idx] = local_data[idx] + remote_data[idx];
    }
}

void symmetric_add_f32(
    torch::Tensor local,
    int64_t remote_ptr,
    torch::Tensor out,
    int64_t n
) {
    const int threads = 256;
    const int blocks = (int)((n + threads - 1) / threads);
    cudaStream_t stream = at::cuda::getCurrentCUDAStream().stream();
    const float* remote = reinterpret_cast<const float*>(static_cast<uintptr_t>(remote_ptr
    ↪ ));

    symmetric_add_kernel<<<blocks, threads, 0, stream>>>(
        local.data_ptr<float>(),
        remote,
        out.data_ptr<float>(),
        n
    );
    C10_CUDA_KERNEL_LAUNCH_CHECK();
}

PYBIND11_MODULE(TORCH_EXTENSION_NAME, m) {
    m.def("symmetric_add_f32", &symmetric_add_f32,
          "UVA symmetric add: local + remote float buffers");
}
'''

_ext = None


def _get_ext():
    global _ext
    if _ext is None:
        _ext = compile_cuda_extension("symmetric_add_uva_ext", CUDA_SRC)
    return _ext
```

```


_symm_cache = None


def _get_symm_state(n: int, dtype: torch.dtype, device: torch.device):
    global _symm_cache
    if _symm_cache is not None:
        c = _symm_cache
        if c["n"] == n and c["dtype"] == dtype:
            return c["buf"], c["hdl"], c["out"]

    buf = symm_mem.empty(n, device=device, dtype=dtype)
    hdl = symm_mem.rendezvous(buf, dist.group.WORLD)
    out = torch.empty(n, device=device, dtype=dtype)
    _symm_cache = {"n": n, "dtype": dtype, "buf": buf, "hdl": hdl, "out": out}
    return buf, hdl, out


@torch.no_grad()
def solution(tensor: torch.Tensor) -> torch.Tensor:
    assert tensor.is_cuda and tensor.is_contiguous()
    assert tensor.dtype == torch.float32
    assert dist.is_initialized()
    assert dist.get_world_size() == 2

    rank = dist.get_rank()
    peer = 1 - rank
    n = tensor.numel()

    if rank == 0:
        _get_ext()
    dist.barrier()

    buf, hdl, out = _get_symm_state(n, tensor.dtype, tensor.device)
    buf.copy_(tensor.reshape(-1))
    hdl.barrier(channel=0)

    remote_ptr = int(hdl.buffer_ptrs[peer])
    _get_ext().symmetric_add_f32(buf, remote_ptr, out, n)

    return out.reshape_as(tensor)
```

Listing 1. Element-wise vector add across two ranks via symmetric memory and using CUDA's unified virtual addressing capabilities. World size must be 2 in this example.

The following prompt structure was used, based on the KernelBench (Ouyang et al., 2025) repository. Note the sections marked in <> are dynamically filled in during evaluation.

```
"""
You write custom CUDA operators to replace **high-level PyTorch and torch.distributed /
    ↪ NCCL-heavy paths** in the given architecture. The reference is a **correctness and
    ↪ API specification**, not a mandate to keep the same operator mix: re-express the
    ↪ workload in the **appropriate backend** (CUDA operators) wherever that unlocks
    ↪ speed.

**Minimize stock PyTorch on the hot path.** Avoid treating off-the-shelf `torch.nn`
    ↪ modules, generic elementwise/tensor ops, or opaque `torch.distributed` collectives
    ↪ as the default implementation when the bottleneck is communication or fused math.
    ↪ Prefer explicit custom kernels, hand-managed buffers, and synchronization you
    ↪ control - only use PyTorch where the harness or interop requires it (e.g., process
    ↪ group setup, tensor shells, loading a compiled extension).

**Prioritize device-side communication.** Favor peer data movement and reductions
```

```
expressed **on device** (symmetric memory, UVA / P2P access, cooperative patterns, CUDA IPC, or backend-native device APIs) instead of repeated host-driven launches of black-box collectives when a tighter device-resident or pipelined pattern fits the problem.

**Maximize compute communication overlap.** Reason about the schedule: chunking, double buffering, multiple streams, pipeline stages, and fused kernels so communication hides behind independent computation (or vice versa). Examples: pipelined allgather  with matmul tiles, chunked reducescatter fused with the next projection, overlapping broadcast with compute on unrelated chunks, ring/tree algorithms implemented where they reduce critical-path latency.

You may replace or fuse collectives, reshape algorithms (ring, tree, recursive halving doubling), or merge communication with adjacent ops. Bias heavily toward **custom CUDA operators**, **device side data movement**, and **overlap** not toward a minimal diff from the reference PyTorch.

Here are examples showing how to embed custom CUDA operators in PyTorch:

Example A: elementwise add (reference vs CUDA extension)
< Example demonstrating baseline with unoverlapped Torch + NCCL >


Optimized with CUDA operators:
< In-context example demonstrating output format with optimized CUDA >

You are given the following PyTorch code to optimize:

< Reference Torch code to optimize >

Note: The kernels should be optimized for BF16 (bfloat16) precision.

Here is some information about the underlying hardware that you should keep in mind.

The GPU that will run the kernel is NVIDIA H100 80GB SXM, Hopper architecture.

- Four GPUs per node; assume peers are NVLink-connected within the same domain. High bisection bandwidth on-node; prefer kernels that exploit tensor cores and async copies.

Here are some concepts about the GPU architecture that could be helpful:

- **Tensor cores**: Use where possible for FP16/BF16/TF32 dense math; fall back to CUDA cores where shapes do not map cleanly.
- **NVLink / P2P**: Device pointers from symmetric memory should be valid for direct loads /stores across peers in this profile.

Here are some best practices for writing kernels on GPU:

- Overlap communication with computation where the schedule allows (e.g., chunked collectives, double-buffering).
- Keep working sets in fast memory hierarchy; minimize cross-GPU traffic for hot loops.

Rewrite `solution()` to get a real speedup by **implementing the workload in the target backend** and **shrinking reliance on stock PyTorch** on the performance-critical path (keep the same signature; must run on every rank like the reference). Use symmetric memory / explicit device buffers and custom compiled code as appropriate.

Output format: You may include a short strategy that states how you exploit device-side communication and compute-communication overlap - then put all implementation in ```python``` fenced blocks: complete `solution()`, supporting kernels, extension/ JIT loading, imports, rendezvous/buffers, etc. No long essays, no tutorial prose, no standalone test harness - only brief strategy (optional) plus working code. If you omit the strategy, the code structure must still reflect overlap and device-side patterns clearly.
```

```
46
47 Produce the complete Python file contents for that path.
48 """
```

Listing 2. System Prompt with task/examples/hardware topology information. This is dynamically filled in during evaluation time from a template.

935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989

B. Survey of GPU Communication Frameworks

Traditional multi-GPU programs separate compute and communication: collective operations (e.g., all-reduce, all-gather) are launched from the host via libraries like NCCL, while compute runs in separate kernels. Fusing these into a single kernel, where communication happens via direct memory loads and stores from within the kernel itself rather than host-launched collectives, is known to yield significant performance gains (Hu et al., 2026a).

PKB is designed around this fused paradigm. Problems require models to move beyond host-side NCCL and instead exploit device-side communication APIs, which collectively enable intra-SM communication: the Torch symmetric heap API for peer-addressable buffers accessible directly from CUDA kernels, NVLink P2P loads and stores via UVA pointers, and NVSwitch multicast via `multimem` PTX instructions. Beyond CUDA, PKB also accepts solutions written in Triton (via the Torch Symmetric Memory API (PyTorch Team, 2026)) and ParallelKittens (Sul et al., 2025), both of which operate within the same intra-node NVLink domain. Whether higher levels of abstraction help or hurt model performance on these tasks is an interesting open question we leave to future work.

For completeness, we briefly survey the inter-node device-side communication landscape, which PKB does not currently cover. NVSHMEM is NVIDIA's native library for inter- and intra-node GPU communication, offering RMA, collectives, atomics, and memory operations; its recent `nvshmem4py` interface and interoperability with Numba-CUDA, CuTe, and Triton have lowered its adoption barrier, though its memory safety features can impose performance overhead (Sul et al., 2025). Triton Distributed similarly spans intra- and inter-node settings, though it primarily targets H800 GPUs. Most recently, NCCL GIN (introduced in NCCL 2.28) provides a device-side API enabling GPUs to directly initiate RDMA transfers without CPU involvement, offering a first-party path for inter-node fused kernels (Hu et al., 2026b).

PKB is intentionally scoped to intra-node settings. This scope excludes certain problem classes that are more naturally inter-node in character — including prefill-decode disaggregation, certain pipeline parallelism patterns, and delta encoding — as these appear more frequently on slower network fabrics (e.g., InfiniBand, RoCE, EFA) where the device-side API landscape is less mature and hardware assumptions are harder to standardize. We view inter-node kernel generation as a natural and important extension of PKB, and welcome contributions in that direction.

990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044

C. Problem Sources

Source repositories.

- `NVIDIA/nccl` (Hu et al., 2026b)
- `stanford-cs336/assignment2-systems` (Stanford CS336, 2026)
- `HazyResearch/ThunderKittens` (Sul et al., 2025; Sul & Re, 2026)
- `NVIDIA/Megatron-LM` (Narayanan et al., 2021)
- `deepspeedai/DeepSpeed` (Jacobs et al., 2023)
- `sands-lab/Global-QSGD` (Xin et al., 2025)
- `deepseek-ai/DeepEP` (Zhao et al., 2025)
- `NVIDIA/TensorRT-LLM` (NVIDIA, b)
- `zhuzilin/ring-flash-attention` (Liu et al., 2023)
- `mlfoundations/open_clip` (Ilharco et al., 2021)
- `NVIDIA/physicsnemo` (PhysicsNeMo Contributors, 2023)
- `nerfstudio-project/gsplat` (Ye et al., 2024)
- `NVIDIA/torch-harmonics` (Bonev et al., 2023)
- `deepmodeling/deepmd-kit` (Wang et al., 2018)
- `pyg-team/pytorch_geometric` (Fey & Lenssen, 2019)
- `dmlc/dgl` (Wang et al., 2019)
- `awslabs/graphstorm` (Zheng et al., 2024a)
- `pytorch/torchrec` (Ivchenko et al., 2022)
- `Zymrael/savanna` (Poli et al., 2023)
- `fla-org/flash-linear-attention` (Yang & Zhang, 2024)
- `hpcaitech/Open-Sora` (Zheng et al., 2024b)
- `SandAI-org/MAGI-1` (ai et al., 2025)
- `facebookresearch/dinov2` (Oquab et al., 2024)
- `facebookresearch/sam3` (Carion et al., 2025)
- `NVIDIA-NeMo/RL` (nem, 2025)
- `NVIDIA-NeMo/emerging-optimizers` (Liu et al., 2025)
- `nv-legate/cupynumeric` (NVIDIA, a)
- `mit-han-lab/distrifuser` (Li et al., 2024)

Methodology. We selected the 87 PKB problems by starting from the parallelism strategies used in production training and inference systems, then choosing representative kernels from the source repositories listed above. For LLM-specific workloads, the majority of the benchmark, production code usually breaks down into composable sharding strategies (tensor, expert, sequence/Ulysses, context/ring, data, FSDP) applied to attention and feedforward layers, with normalization between stages. Each path through this taxonomy (Figure 1) creates optimization opportunities both inside layers and at the transition points where data moves between parallel layouts. These boundary kernels are especially important because communication is not just a separate collective. It is part of the operation itself, like in MoE routing, vocab-parallel log-prob computation, and ring attention. Thus, PKB includes many variable-size and irregular communication patterns. Beyond LLMs, we include 22 problems from other domains, summarized in Table 4. We also group all PKB problems into six abstraction tiers in Table 3: the first tier is bare collectives, and the other tiers combine computation and communication at higher levels of complexity.

Tier	Count	Description
Bare collective primitives	8	Fundamental communication operations without any added computation. These serve as a baseline for the raw data-transfer speed of the system.
Transition kernels	8	Simple mathematical operations—like RMSNorm or data compression (FP8 quantization)—that are fused directly into communication steps to reduce overhead.
Layer fragments	18	Building blocks of neural network layers, such as GEMM or convolutions, that have been split across multiple GPUs using standard parallelism strategies.
End-to-end composites	16	Complex, multi-step operations that coordinate both calculation and data movement, such as full Ring Attention or MoE routing and dispatch.
Optimizer / update kernels	9	Operations used during the model update phase, where gradients and parameters are synchronized and updated across a cluster (e.g., ZeRO-powered sharded optimizers).
Distributed loss functions	5	Final-stage calculations that compare model outputs to targets across all GPUs, such as cross-entropy for very large vocabularies or contrastive losses for vision models.
Application-specific kernels	20	Specialized workloads tailored for non-LLM domains like GNNs, Scientific simulations, and 3D Graphics, as detailed in Table 4.

Table 3. Abstraction tiers covered by PKB.

Coverage beyond standard LLM kernels. Figure 3 summarizes coverage by parallelism axis and domain; Table 4 describes what the non-LLM problems contribute that standard transformer problems don't. We under-emphasize pipeline parallelism (1 problem) since PP performance is in practice mostly inter-node and governed by scheduling, and we exclude prefill/decode disaggregation as primarily an inference-serving scheduling concern (not a kernel concern).

Block	Problems	What it contributes
GNN	65–70	Graph workloads present ownership-based routing: each rank owns different nodes or features, so data must be fetched, pushed, deduplicated, or scored across GPUs. This creates variable-size all-to-all communication and load imbalance from uneven graph neighborhoods, making GNN parallelism closer to sparse routing than dense tensor parallelism.
Scientific	60–64	Scientific kernels add non-transformer communication patterns: distributed FFT transposes, spherical convolutions over a spatial process grid, shared reductions inside an optimizer, and 3D Gaussian splatting where cameras and projected points must be redistributed. These problems stress emphasize layouts rather than token or expert layouts.
Vision & diffusion	59, 77–82, 88	Vision and diffusion models introduce communication over images, tiles, masks, and feature databases. Examples include ring contrastive losses, async attention with all-to-all exchange, tile-parallel VAE decoding, distributed k-NN, Sinkhorn assignment, mask-IoU suppression, and Conv2d boundary exchange.
Recommendation / sparse	10, 71	These workloads deal with ”jagged” data (like variable-length lists of user interests). Because the data is not in neat, equal-sized blocks, the system is forced to move lists of varying lengths, which is a major stress test for data routing.

Table 4. Non-LLM problem categories and what they contribute.

D. Communication-Aware Roofline

The classical roofline bounds runtime by peak compute P_{peak} and HBM bandwidth B_{mem} (Williams et al., 2009). For distributed kernels, we add NVLink bandwidth B_{comm} as a third ceiling, since it is often the bottleneck. A naive extension takes the three-way maximum across all three resources, but this overstates the achievable speedup whenever a workload contains a barrier that no implementation can hide. We therefore use a phase-aware roofline where communication is a third resource and we serialize only at unavoidable dependencies.

D.1. Phases as dependency boundaries

Fused kernels can overlap compute, HBM traffic, and communication (e.g. a fast GEMM–all-gather implementation streams weights into the matmul units while later tiles are still in flight on the network). We model these patterns with a per-resource max. However, some operations enforce hard dependencies: for example, an MoE routing all-to-all must finish before expert compute begins. We model these with a sum.

We decompose each problem into phases within which the three resources overlap freely. Within a phase, we group operations by tile-pipelining ability: anything that can stream from one operation's output into another's input at sub-tensor granularity belongs to the same phase, and anything requiring a fully formed intermediate sits across a boundary. We also report a coarse non-phase-aware roofline per problem.

D.2. The roofline equation

For problem i split into phases k with FLOPs W_k, HBM traffic $Q_{\mathrm{mem},k}$, and communication volume $Q_{\mathrm{comm},k}$, the phase-aware roofline time is

$$T_{\mathrm{roof},i} \;=\; \sum_k \max\left(\frac{W_k}{P_{\mathrm{peak}}},\; \frac{Q_{\mathrm{mem},k}}{B_{\mathrm{mem}}},\; \frac{Q_{\mathrm{comm},k}}{B_{\mathrm{comm}}}\right). \tag{3}$$

The three peaks $P_{\mathrm{peak}}, B_{\mathrm{mem}}, B_{\mathrm{comm}}$ come from the NVIDIA H100 SXM5 specs, and the three numerators come from a maximally fused implementation of each stage.

Compute. P_{peak} is set per stage by precision and execution path. For H100 SXM5 we use dense, non-sparse peaks: 989 TFLOP/s (BF16/FP16 Tensor Core), 1979 TFLOP/s (FP8 Tensor Core), 494 TFLOP/s (TF32 Tensor Core), 67 TFLOP/s (FP32 SM cores).

HBM. $B_{\text{mem}} = 3.35$ TB/s. $Q_{\text{mem},k}$ counts bytes crossing the L2–HBM boundary; register- and SRAM-resident intermediates are free.

Communication. $B_{\text{comm}} = 450$ GB/s per direction. $Q_{\text{comm},k}$ uses collective bus-volume formulas (e.g. $\frac{W-1}{W}M$ for a ring all-gather of M bytes across W ranks). When a phase spans multiple process-group axes (e.g. TP and CP), each stage is charged on its own axis, and times sharing a network resource are summed before the phase max.

D.3. Cost models

Per problem, we contribute a Python cost-model file that declares its phases, per-stage FLOP / HBM / communication formulas, and the symbolic shape variables they depend on (like world size, sequence length, expert count). To evaluate, we bind the shape variables to a hardware profile supplying P_{peak}, B_{mem}, and B_{comm}. The same model produces T_{roof} at any shape or world size.

D.4. Worked example: Problem 72 Hyena Conv1D Boundary Exchange

This problem is a depthwise 1D conv with zigzag context-parallel sharding. Each rank owns chunks of shape $[B, H, 2S]$ and filter $[H, 1, K]$, exchanges $K-1$-element halos with both neighbors, then runs a local 1D conv. Halos stream from P2P receive buffers into registers while the convolution computes, so the workload is one phase with two stages:

- Halo P2P: $W = 0,\ Q_{\text{mem}} = 0,\ Q_{\text{comm}} = 2BH(K-1)\,d_b$.
- Depthwise Conv1D: $W = 4BHSK,\ Q_{\text{mem}} = (4BHS + HK)\,d_b,\ Q_{\text{comm}} = 0$.

At $(B, H, K, S, W, d_b) = (1, 1024, 7, 1024, 4, 2)$ the roofline gives $W/P_{\text{peak}} = 438$ ns, $Q_{\text{mem}}/B_{\text{mem}} = 2{,}508$ ns, and $Q_{\text{comm}}/B_{\text{comm}} = 55$ ns, so $T_{\text{roof},72} \approx 2.51\,\mu$s (memory-bound).

D.5. Baseline utilization across PKB

We classify each problem's binding resource as whichever of compute, HBM, or NVLink contributes the largest share of $T_{\text{roof},i}$ summed across phases. Of the PKB problems, 75% are NVLink-bound, 18% are HBM-bound, and 7% are compute-bound, confirming that the benchmark is dominated by communication-limited workloads and motivating the third ceiling in Equation 3.

Stratified utilization. We compute the PyTorch reference utilization $U_i = \min(1,\ T_{\text{roof},i}/T_{\text{ref},i})$ on every task and stratify by binding resource:

Binding	N	$\%\,U_{\text{ref}} < 0.5$
NVLink (communication)	65	100%
HBM (memory)	16	100%
Compute	6	83.3%

Table 5. PyTorch reference utilization, stratified by the binding resource of each problem's roofline. Every PyTorch baseline on the 81 communication- and memory-bound tasks (93% of PKB) falls below half its hardware ceiling.

1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264

E. Benchmark Variance

Latency Measurement Methodology. For each kernel we conduct $T=5$ independent evaluation trials under distinct RNG seeds and full harness restarts. Within each trial, both the reference and CUDA implementations execute a fixed 500 iteration warmup phase followed by $P=100$ timed profiling iterations per rank; the reported per-trial latency is the total wall time divided by P, taken as the maximum across ranks from the per-rank performance logs. Bars in Figure 9 report the mean of these T per-trial latencies; error bars denote ± 1 sample standard deviation across trials (computed independently for the reference and CUDA backends). This performance benchmarking design follows the established practices from ThunderKittens 2.0 (Sul & Re, 2026).

As Figure 9 shows, the large majority of problems exhibit tight error bars, confirming that the measurement pipeline is stable. A small number of higher-latency kernels display wider spread, but the spread appears to be comparable between the reference and CUDA implementations for any given problem, indicating that the observed variance reflects system-level noise rather than instability in a particular generated solution.

F. Agentic Harness

In addition to single-shot prompting (Appendix A), we evaluate models in an agentic setting using mini-swe-agent (Yang et al., 2024). The agent is given the same kernel specification as the single-shot setting, but instead of producing a solution in one pass, it operates in a loop: the LLM observes its current context, decides which shell command to run (e.g., writing code to a file, executing the evaluation script, reading error output), and iterates based on the result. Concretely, the agent is provided with an evaluation command it can invoke at any point to run its current solution against the correctness and performance harness on real hardware — giving it the same feedback signal a human developer would have. The loop terminates when the agent is satisfied with its solution or a turn budget is exhausted. An illustration is provided in Figure 10. The agent was able to correctly solve 31 problems, with 26 solutions being faster than the reference.

G. Kernels of Interest

Here, we highlight a selection of generated kernels that exhibit effective multi-GPU optimization strategies. We highlight kernels that outperform strong baselines, as well as those that represent genuinely novel contributions as of writing.

G.1. NCCL Collectives (Gemini 3 Pro)

We observe several generated kernels outperform NCCL collectives outright at low tensor sizes. Gemini 3 Pro (single shot) produced an AllGather kernel running 1.2× faster than the NCCL baseline on tensor shape (1024×1024), and a Scatter kernel running 1.626× faster on Rank 0 tensor shape $4 \times 1024 \times 1024$ (world size is 4). Performance matches NCCL at smaller matrix sizes (Figure 11), but degrades noticeably as matrix size increases. One key optimization the model independently identified was dynamically selecting the widest available memory access width based on buffer alignment at runtime (Figure 12).

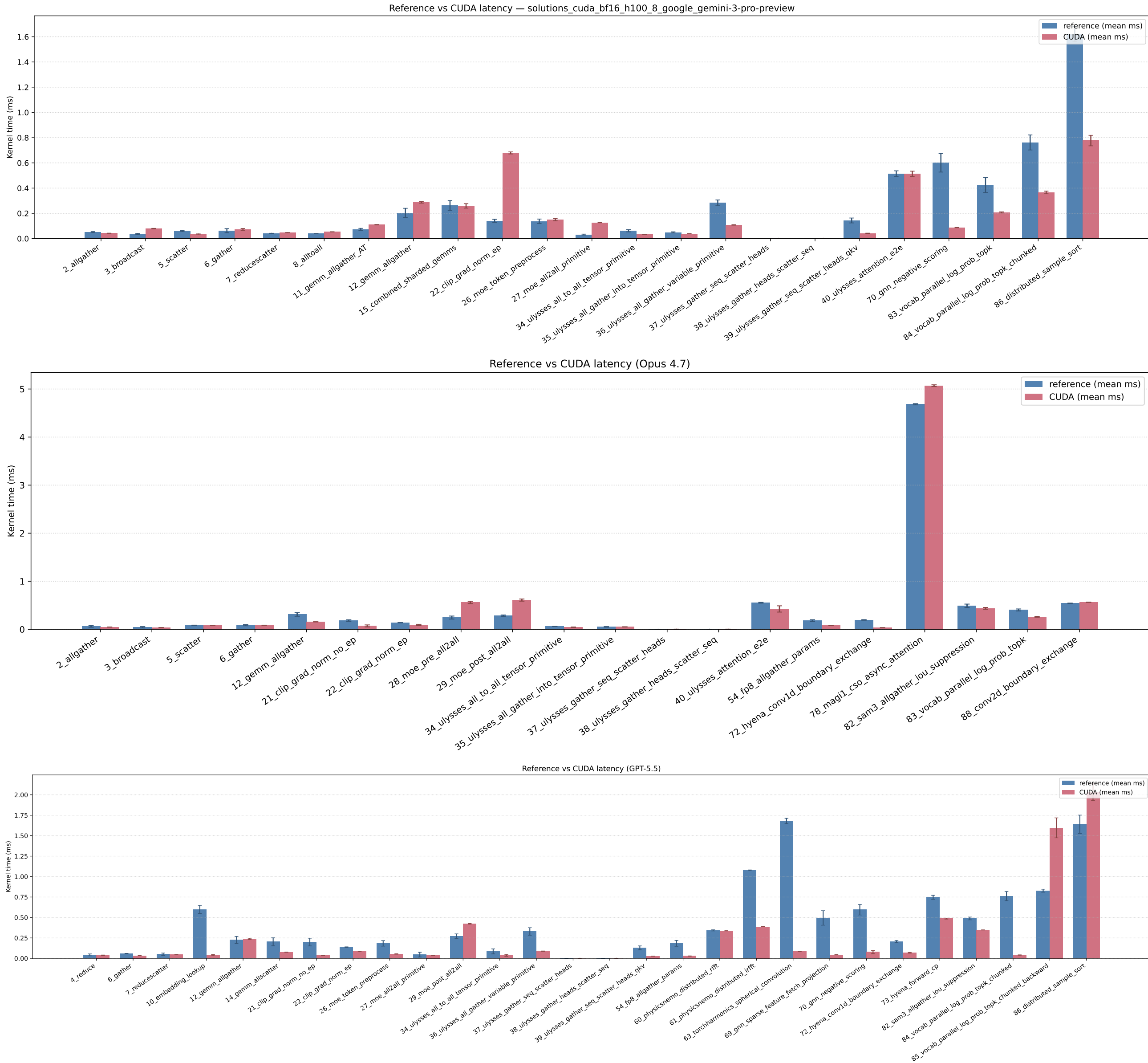


Figure 9. **Per-problem kernel latency and run-to-run variance for Gemini 3 Pro, Claude Opus 4.7, and GPT-5.5 (pass@1).** Each trial internally averages P=100 profiling iterations. Bars show mean $\pm$ 1 SD across T=5 independent evaluation trials for the reference implementation (blue) and each CUDA solution (red). The Gemini 3 Pro plot excludes `29_moe_post_all2all`, whose latency is an outlier that would otherwise compress the scale, it does not exhibit anomalous variance.

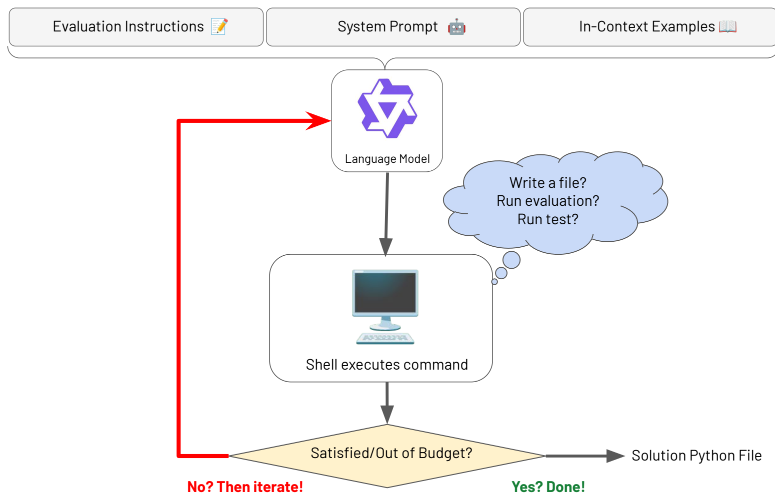


Figure 10. **Flow of agent harness**. The LLM can make arbitrary modifications to an existing codebase based on the output of the provided evaluation script. Test-time compute is set to a limit of 20 iterations, i.e. LLM calls. The harness used is mini-swe-agent (Yang et al., 2024).

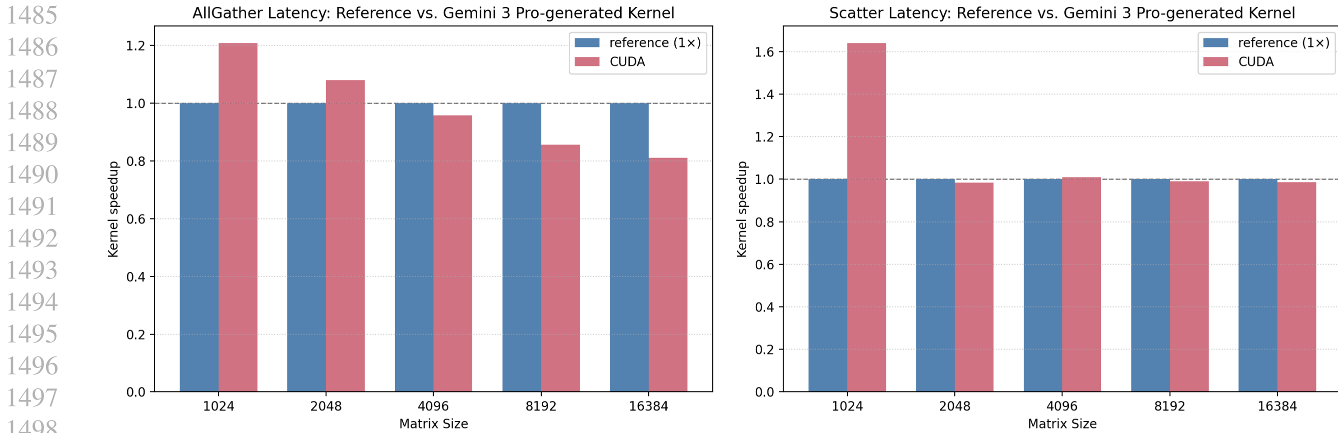


Figure 11. **Generated symmetric-memory kernels can approach NCCL performance on Scatter and AllGather collectives..** "Reference" in this case refers to the `torch.distributed` NCCL collective. The model evaluated is Gemini 3 Pro (Single-Shot).

```
// Dynamically use the widest memory instructions possible based on alignment
if (chunk_bytes % 16 == 0) {
    int64_t chunk_16 = chunk_bytes / 16;
    const uint4* src_16 = reinterpret_cast<const uint4*>(src);
    uint4* dst_16 = reinterpret_cast<uint4*>(dst_ptr);
    for (int64_t i = idx; i < chunk_16; i += stride) {
        dst_16[i] = src_16[i];
    }
} else if (chunk_bytes % 8 == 0) {
    int64_t chunk_8 = chunk_bytes / 8;
    const uint2* src_8 = reinterpret_cast<const uint2*>(src);
    uint2* dst_8 = reinterpret_cast<uint2*>(dst_ptr);
    for (int64_t i = idx; i < chunk_8; i += stride) {
        dst_8[i] = src_8[i];
    }
}
```

Figure 12. **Gemini 3 Pro Key Optimization: vectorized memory copy kernel.** To optimize an NCCL AllGather operation, the model dynamically selects the widest available memory instruction based on buffer alignment: 128-bit `uint4` loads/stores when the chunk size is 16-byte aligned, falling back to 64-bit `uint2` otherwise. This maximizes memory throughput on the GPU.

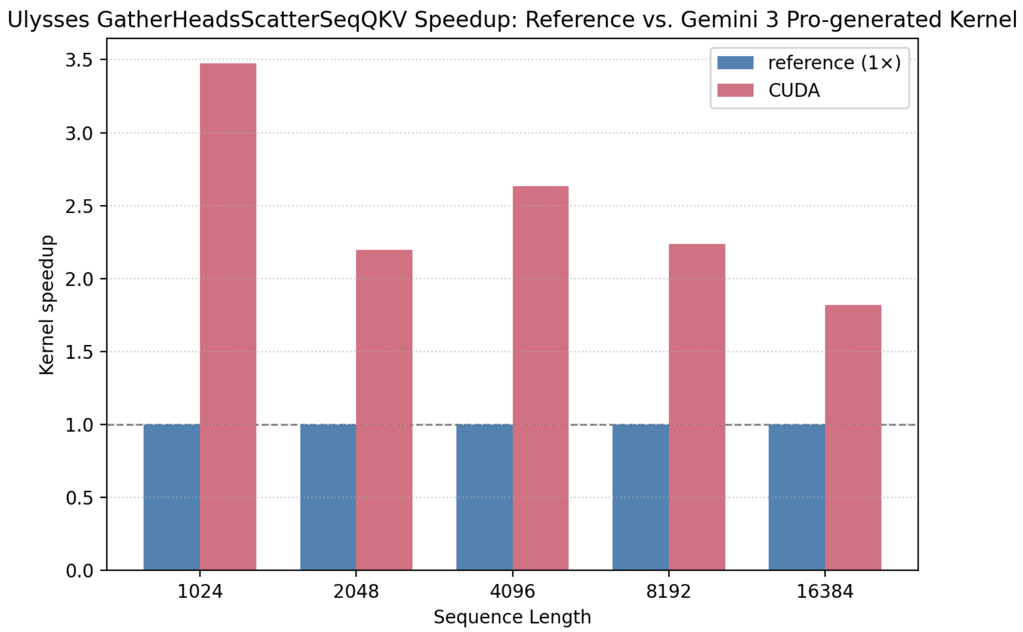
G.2. Context Parallel: Ulysses (Gemini 3 Pro)

Sequence parallelism via Ulysses Attention redistributes a tensor from sequence-parallel to head-parallel layout before the attention computation, then back again afterward (Jacobs et al., 2023). In the reference implementation, this is performed as a gather over the sequence dimension followed by a scatter over the head dimension, both via `all_to_all` calls interleaved with reshapes, transposes, and concatenations—inducing substantial overhead.

Models perform surprisingly well on this class of problems, averaging a 28.3% solve rate in the PKB testing harness (Table 2). The best generated solution—produced by Gemini 3 Pro—achieves a $3.44\times$ speedup over the PyTorch + NCCL baseline on the `GatherHeadsScatterSeqQKV` kernel at sequence length $T = 1024$, with input tensors of shape $(B, T, 3 \times H \times d) = (1, 1024, 3072)$, where B is the batch size, $H = 16$ is the number of attention heads, and $d = 64$ is the per-head dimension, packed as a single QKV projection tensor. Speedups across sequence lengths are shown in Figure 13.

The key insight Gemini 3 Pro identified is that the entire `all_to_all` can be recast as a direct NVLink *pull* kernel over symmetric memory (Figure 14), eliminating host-orchestrated data movement entirely. The reference implementation incurs multiple sequential memory passes (split, all-to-all, concat, reshape), whereas the generated kernel fuses these into two operations per rank: a single contiguous write of local data into a symmetric buffer, followed by a single NVLink read in which each rank directly pulls the slices it needs from its peers into the correct output layout. Destination offsets are computed analytically via an index decomposition inside the kernel, eliminating all intermediate allocations and host-side synchronization points.

1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630



1631
1632
1633

Figure 13. **Ulysses `GatherHeadsScatterSeqQKV` speedup** of the Gemini 3 Pro-generated CUDA kernel relative to the PyTorch + NCCL reference ($1\times$) across sequence lengths. The generated kernel achieves up to $3.44\times$ speedup at T=1024, sustaining $> 1.8\times$ across all evaluated sequence lengths.

1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649

```
__global__ void ulysses_pull_kernel(
    const uint64_t* __restrict__ peer_ptrs,
    void* __restrict__ out_ptr,
    int64_t A, int64_t B, int64_t C, int64_t D, int64_t E_vec,
    int P, int my_rank, bool scatter_first, int64_t N_out_vec
) {
    int64_t idx = (int64_t)blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= N_out_vec) return;

    int64_t e = idx % E_vec;
    int64_t tmp = idx / E_vec;

    int64_t rank_j;
    int64_t in_idx;

    if (scatter_first) {
        // B is scatter (head_dim), D is gather (seq_dim)
        // Output shape: [A, B/P, C, D*P, E_vec]
        int64_t d_out = tmp % (D * P);
        tmp = tmp / (D * P);
        int64_t c = tmp % C;
        tmp = tmp / C;
        int64_t b_out = tmp % (B / P);
        int64_t a = tmp / (B / P);

        rank_j = d_out / D;
        int64_t d_in = d_out % D;
        int64_t b_in = my_rank * (B / P) + b_out;

        in_idx = (((a * B + b_in) * C + c) * D + d_in) * E_vec + e;
    }
}
```

Figure 14. **NVLink pull kernel generated by Gemini 3 Pro.** Rather than staging data through intermediate buffers as in a conventional AllToAll, each thread directly computes its source peer and remote index, then pulls the required element over NVLink. The `scatter_first` flag controls whether the scatter dimension is the head dimension D or the sequence dimension B, handling both Ulysses gather-scatter orderings in a single kernel and eliminating all packing and staging overhead.

G.3. NeMo Vocab-Parallel LogP with Top-k/Top-p Filtering (Gemini 3 Pro)

We identified a kernel in NVIDIA NeMo/RL that computes per-token log-probabilities of target tokens under a top-k/top-p filtered distribution, where logits are sharded across the vocabulary dimension over a tensor-parallel group. The reference uses `all_to_all_single` to transpose the layout from vocab-parallel to sequence-parallel, applies top-k/top-p filtering via a sort and cumulative-sum mask, computes `F.log_softmax`, gathers the target log-probabilities, and finally `all_gather`s the scalar results across ranks.

An illustrative comparison of the reference and generated solutions is provided in Figure 15. We find the optimized solution by Gemini 3 Pro to be $2.05\times$ faster on input (batch=4, seq_len=2048, vocab_size=1024, 256 vocab entries per rank across 4 ranks) with top-k=10 and top-p=0.9. Speedups over different sequence lengths is shown in Figure 16. The kernel appears to have three main advantages:

1. It replaces `all_to_all_single` with `push_logits_kernel_vec`, a custom kernel that writes logits directly into peer GPU symmetric memory buffers using vectorized 128-bit (`uint4`) stores, while performing the vocab-to-sequence layout permutation implicitly in the index arithmetic, eliminating NCCL staging overhead entirely.

2. It fuses log-softmax, target token extraction, and the final all-gather into a single block-per-token kernel (`fused_log_softmax_gather_push`), which performs a two-pass warp-shuffle reduction (max, then sum) over the full vocabulary in shared memory and directly writes each scalar log-probability into all peer `logprobs_buf` buffers in one shot, replacing `dist.all_gather`.

3. The sequence is chunked and processed iteratively, so the symmetric-memory barrier after the push blocks only on a fraction of the data at a time, reducing the exposed synchronization latency compared to the single global barrier implicit in `all_to_all_single`.

PyTorch Reference:

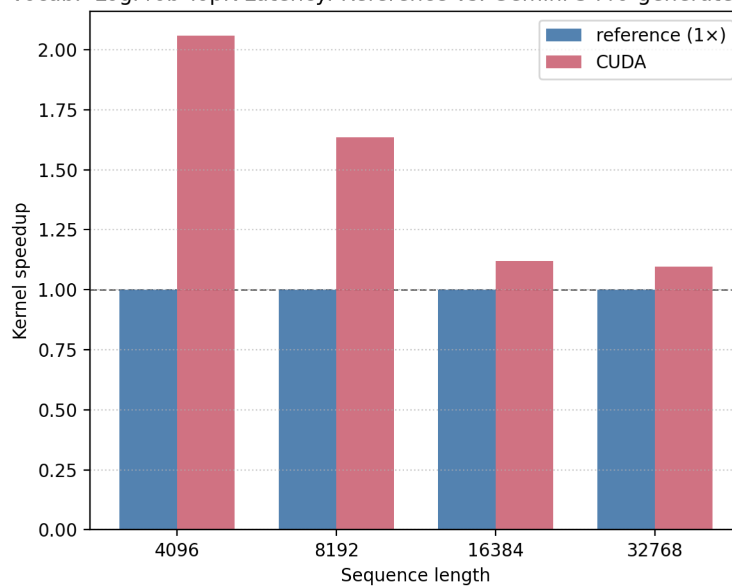
```
# [1] Transpose layout:
#     vocab-parallel -> seq-parallel
logits = all_to_all_single(logits_2d)
# [2] Filter + score target token
logits = apply_top_k_top_p(logits)
log_probs = F.log_softmax(logits)
token_lp = torch.gather(
    log_probs, target)
# [3] Replicate scalar log-probs
dist.all_gather(gathered, token_lp)
```

LLM-Generated Solution:

```
for start in range(0, num_tokens,
                   chunk_tokens):
  # Replaces [1]: vectorized peer push
  # uint4 stores + implicit permute
  __global__ push_logits_kernel_vec():
    dst[t_local * W*V + rank * V]
      = src_vec[i]  // 128-bit store
  recv_hdl.barrier()
  # [2] unchanged: top-k/p filter
  filtered = apply_top_k_top_p(...)
  # Replaces [1]+[3]: fused kernel
  __global__ fused_log_softmax_
             gather_push():
    // warp-shuffle max+sum reduction
    lp = target_val-(max+log(sum))
    for dst in range(world_size):
      peer_buf[dst][token] = lp
  logprobs_hdl.barrier()
```

Figure 15. **Gemini 3 Pro can optimize a vocabulary-parallel log-probability kernel via symmetric memory and fused reductions.** The reference transposes logits from vocab-parallel to sequence-parallel layout via `all_to_all_single`, applies top-k/top-p filtering, computes `log_softmax`, and replicates scalar results via `all_gather`. The LLM-generated solution replaces both collectives: a vectorized push kernel writes logits directly into peer symmetric memory buffers (with the layout permutation implicit in the index arithmetic), and a fused block-per-token kernel performs warp-shuffle softmax reduction and writes each scalar log-probability directly to all peer buffers, eliminating NCCL staging entirely. Processing is chunked to reduce barrier exposure.

1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850



1851
1852
1853

Figure 16. **Gemini 3 Pro-generated VocabP LogProb TopK kernel outperforms public reference.** The solution detailed in Figure 15 outperforms the reference implementation across a sweep of tensor sizes in the PKB testing harness (nem, 2025), with correctness verified at each configuration.

1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869

G.4. Hyena Forward CP (GPT-5.5)

This kernel implements a section of the **Hyena context-parallel forward pass** from the official repository (Poli et al., 2023). Each rank starts with sequence-sharded activations x_1, x_2, v of shape $[B, D, l]$, where B is the batch size, D is the model hidden dimension, and $l = L/N$ is the local sequence length for global sequence length L distributed across N GPUs.

The generated kernel achieves a $1.53\times$ speedup on the benchmark configuration (sequence tensors of shape $[1, 1024, 2048]$, filter $[1024, 4096]$, N=4 GPUs) by replacing two `all_to_all_single` round-trips with direct UVA peer reads/writes, eliminating intermediate materialized tensors and fusing gating, bias, and zigzag reindexing into the data movement passes (Figure 18). However, as shown in Figure 17, the speedup does not hold at longer sequence lengths: the kernel is $1.72\times$ faster at l=2048 but slows to $0.87\times$ and $0.76\times$ at l=4096 and l=8192 respectively, suggesting that the UVA peer-read approach has other limitations as the working set grows.

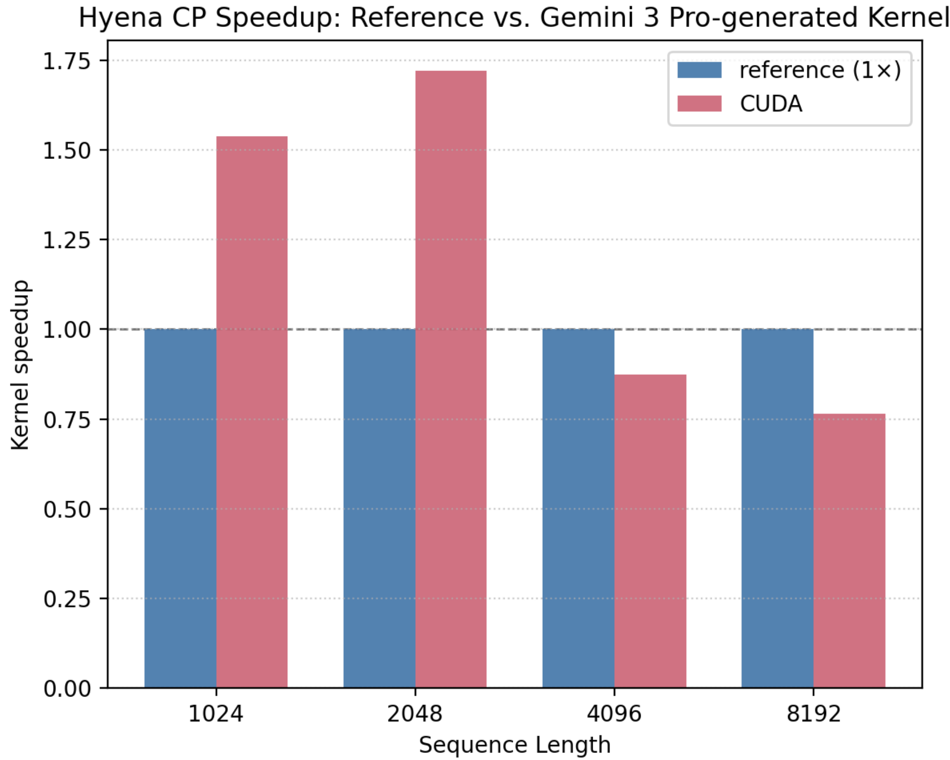


Figure 17. **Gemini 3 Pro-generated Hyena CP kernel outperforms the public reference at short sequence lengths but degrades at longer ones.** Speedup over the reference implementation (Poli et al., 2023) across local sequence lengths of 1024–8192 on 4 GPUs. The generated kernel achieves up to $1.71\times$ speedup at l=2048 but falls below the reference ($0.87\times$) at l=4096.

PyTorch Reference:

```
# [1] Seq-sharded -> channel-sharded
#     (all_to_all x3, one per tensor)
x1 = _a2a_split_to_full(x1_seq, ...)
x2 = _a2a_split_to_full(x2_seq, ...)
v  = _a2a_split_to_full(v_seq,  ...)
# [2] Local FFT convolution
z = x2 * v
z = _fftconv_ref(z, h_local,
                 bias_local)
z = x1 * z
# [3] Channel-sharded -> seq-sharded
#     (all_to_all + zigzag reindex)
z_seq = _a2a_full_to_split(
    z, group, with_zigzag)
```

LLM-Generated Solution:

```
# Replaces [1]: pack all three into
# one symmetric buffer, one barrier
pack3_bf16(x1, x2, v, inp_symm)
inp_hdl.barrier()
# Replaces [1]+[2] partially: UVA
# peer reads fused with x2*v and
# inverse zigzag reindex in one kernel
__global__ gather_x1_and_u():
  pre_t = inv_zigzag_chunk(ch, W)
  u[idx] = bf16(x2[peer]) * bf16(v[peer])
  x1_full[idx] = x1[peer]
# [2] unchanged: FFT convolution
rfft -> complex_filter_mul -> irfft
# Replaces [2]+[3]: bias, x1*z,
# and scatter to [B,l,D] in one kernel
__global__ finalize_z + scatter_bsl():
  conv = y[t] + u[t] * bias[c]
  z_symm[t] = x1_full[t] * conv
z_hdl.barrier()  // then UVA scatter
```

Figure 18. **GPT-5.5 eliminates redundant all-to-all collectives in Hyena context parallelism via fused symmetric-memory kernels.** The reference performs three separate `all_to_all` transposes (one per input tensor) to go from sequence-sharded to channel-sharded layout, runs the local FFT convolution, then transposes back. The LLM-generated solution packs all three inputs into a single symmetric allocation and replaces the first round of collectives with a single UVA peer-read kernel (`gather_x1_and_u`) that simultaneously computes $x_2 \cdot v$ and applies the inverse zigzag reindexing inline. The final `all_to_all` back is replaced by `scatter_final_bsl`, which fuses the residual bias, $x_1 \cdot z$ multiply, and transpose to $[B, l, D]$ into a single device-side scatter.

G.5. SAM3 all-gathered mask IoU suppression (GPT-5.5)

This kernel implements **SAM3 all-gathered mask IoU suppression** for distributed video object segmentation, adapted from the official SAM3 repository (Carion et al., 2025). Each rank owns a variable number of tracked object masks and their object scores. The reference implementation uses two `dist.all_gather` calls (one for masks, one for scores), and computes pairwise IoU scores between all gathered masks in nested loops before applying a filter that removes masks above a fixed IoU threshold.

The generated kernel achieves a $1.40\times$ speedup at the benchmark configuration: 256×256 masks with 1–3 tracked objects per rank across 4 GPUs (4–12 total objects) and an IoU threshold of 0.7. The speedup comes from collapsing this pipeline into four CUDA kernels backed by symmetric memory (Figure 20): each rank packs its local masks and scores into a symmetric buffer at the offset corresponding to its global object slice. However, as shown in Figure 19, the speedup does not generalize to larger mask resolutions: the kernel slows to $0.41\times$ at 512×512 and $0.15\times$ at 1024×1024 (where NCCL's all-gather overhead is offset by its ability to more effectively saturates NVLink bandwidth compared to UVA peer reads).

2035 2036 2037 2038 2039 2040 2041 2042 2043 2044 2045 2046 2047 2048 2049 2050 2051 2052 2053 2054 2055 2056 2057 2058 2059 2060 2061 2062 2063 2064 2065 2066 2067 2068 2069 2070 2071 2072 2073 2074 2075 2076 2077 2078 2079 2080 2081 2082 2083 2084 2085 2086 2087 2088 2089

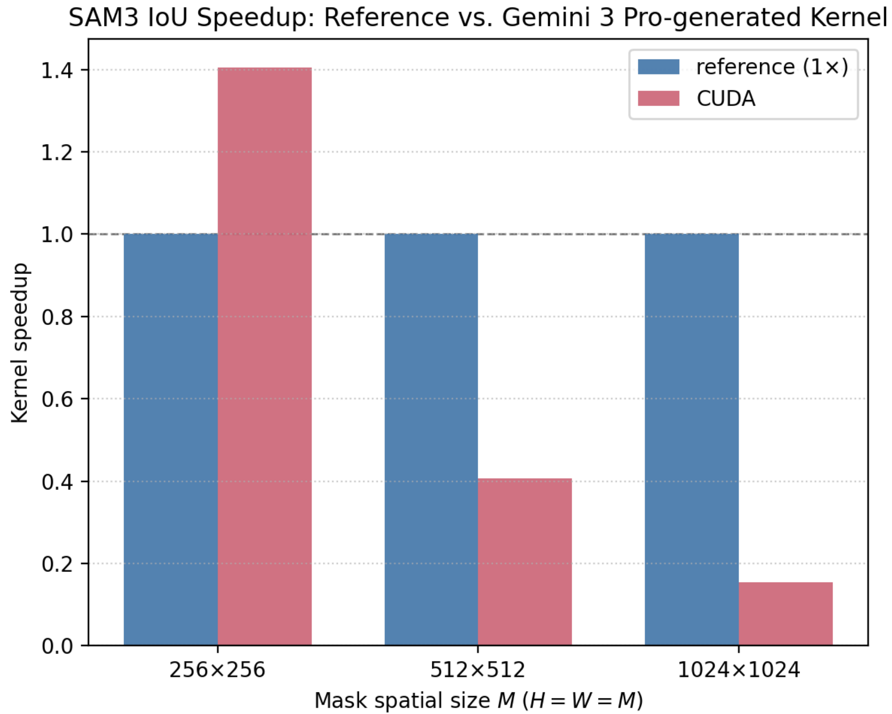


Figure 19. **Gemini 3 Pro-generated SAM3 IoU kernel outperforms the public reference at small mask resolutions but degrades sharply at larger ones.** Speedup over the reference implementation (Carion et al., 2025) across square mask resolutions $M \times M$ on 4 GPUs with 1–3 objects per rank. The generated kernel achieves $1.40\times$ speedup at $M{=}256$ but falls to $0.41\times$ at $M{=}512$ and $0.15\times$ at $M{=}1024$, indicating that the symmetric-memory + bitset approach wins on per-call overhead at small sizes but loses to NCCL's bandwidth-tuned all-gather as the working set grows.

PyTorch Reference:

```
# [1] Two variable all-gathers
masks = _all_gather_variable(
    masks_local, counts, group)
scores = _all_gather_variable(
    scores_local, counts, group)
# [2] Dense IoU matmul on float masks
binary = masks > 0
flat = binary.flatten(1).float()
inter = flat @ flat.T  # N x HW x N
area = flat.sum(dim=1)
union = area[:,None]+area[None,:]-inter
iou = inter / union.clamp_min(1.0)
# [3] Suppression via broadcast compare
overlaps = triu(iou >= thresh, 1)
suppress = (overlaps & cmp(last_i,
            last_j) & (last_j > -1))...
masks[suppress] = _NO_OBJ_LOGIT
```

LLM-Generated Solution:

```
# Replaces [1]: pack into symmetric
# buffer at global offset, one barrier
pack_local_to_sym(masks, scores,
    sym_flat, global_offset)
hdl.barrier()
# Replaces [1]+[2] partially: UVA peer
# reads fused with bitpack + area
__global__ gather_bitpack_area():
  dst[p] = peer_base[obj * pixels + p]
  bits |= (dst[p] > 0) << b  // 32/word
  area = popcount-reduce(bits)
# Replaces [2]+[3]: bitset IoU
__global__ pair_suppress_bitset():
  for w in words:
    inter += __popc(bi[w] & bj[w])
  if inter >= thresh * union:
    suppress[i or j] = 1
# Replaces in-place fill
__global__ apply_suppression():
  if suppress[obj]:
    masks_out[idx] = -10.0f
```

Figure 20. **LLM eliminates dense IoU matmul and dual all-gathers via bitpacked symmetric-memory kernels.** The reference performs two `dist.all_gather` calls and computes pairwise IoU as a dense $N \times HW \times N$ matmul over float-binarized masks. The LLM-generated solution packs both masks and scores into a single symmetric buffer (replacing both all-gathers with one barrier), then a fused gather kernel reads peer memory over UVA pointers while simultaneously bitpacking each mask to 32-bit words and computing the per-object area via popcount reduction.

H. Future Work

PKB focuses on intra-node NVLink topologies and deliberately excludes problems where performance is governed by scheduling rather than kernel design, such as pipeline parallelism and prefill-decode disaggregation. Evaluations are conducted on H100 with NVSwitch; extending coverage to other topologies is left to future work. Future work may also leverage more specialized agentic systems and reasoning techniques to improve performance. PKB has natural support for ParallelKittens and Triton, and studying the impact of different programming abstractions — alongside emerging interfaces like cuTile's upcoming networking features, NCCL GIN, and NVSHMEM — would be valuable follow-on work. Finally, extending the benchmark to internode communication via RoCE and InfiniBand is a natural next step, in addition to potentially expanding to other accelerators and topologies such as TPUs. We hope that our work inspires development and progress in this domain.

I. Ethics Statement

This paper presents novel kernel optimization techniques for distributed systems, demonstrating measurable reductions in energy consumption across large-scale AI data centers. By minimizing computational overhead at the kernel level, these methods directly reduce the carbon footprint of AI infrastructure — a critical step toward sustainable, large-scale machine learning. That said, the complexity of deploying such optimizations across heterogeneous hardware environments may introduce implementation overhead from running LLMs, and efficiency gains could inadvertently encourage greater computational scale rather than reduction in energy use.