KERNELMOSAIC: AN OPEN DATASET AND AGENTIC FRAMEWORK FOR CUDA KERNEL GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Training large language models to generate efficient CUDA kernels requires supervision spanning diverse computational requirements and iterative code optimization. Constructing this supervision is challenging because of heterogeneous specifications, and useful implementations emerge through iterative complicated testing and refinement. We introduce KernelMosaic, an open dataset pairing diverse CUDA tasks with verified solutions and trajectories. Human-guided review and diversity-aware selection curate tasks expressed as natural-language descriptions, reference programs, mathematical specifications, and their intersections. To build the solutions and trajectories, our agentic framework guides a strong LLM through implementation, correctness repair, and performance optimization with reusable CUDA guidance and execution feedback. Supervised fine-tuning of a small open-source model on KernelMosaic raises correctness from 4%, 0%, and 0% to 88%, 82%, and 16% on KernelBench L1, L2, and L3, respectively, while increasing the fraction of tasks with kernels faster than PyTorch eager at every level. Further training experiments show that supervision from the curated trajectories substantially outperforms direct responses and promotes sustained testing and revision. These results demonstrate that KernelMosaic enables a general-purpose model to achieve substantial gains in CUDA kernel generation.

1 INTRODUCTION

Efficient GPU kernels connect mathematical computation to the hardware that executes it. Developing them requires reconciling numerical requirements with parallel execution, memory access, and software interfaces. Automating this work with large language models requires reusable supervision covering diverse computational tasks and the iterative decisions that produce correct and efficient implementations.

Existing resources demonstrate several approaches to kernel-generation supervision: KernelBook provides paired implementations (Paliskara & Saroufim, 2025), ConCuR emphasizes concise reasoning (Kong et al., 2025), and CUDA Agent and Nemotron-SFT-CUDA-v1 use agentic execution to develop solutions (Dai et al., 2026; NVIDIA, 2026). Constructing supervision across independently sourced natural-language, programmatic, and mathematical tasks introduces a joint curation and generation problem. Specifications differ in completeness and computational content, while their solutions require different evaluation procedures. A reusable collection must therefore preserve computational intent, maintain task diversity, and support the generation of tested implementations and informative trajectories.

Two challenges follow. First, task curation must resolve missing context and reduce redundancy without discarding distinct computations or changing their meaning. Second, solution generation must produce both reliable implementations and useful supervision of the iterative process. Correctness repair and performance optimization involve successive attempts; recording these attempts together with their feedback captures how implementation choices evolve into a working solution.

To address these challenges, we introduce KernelMosaic, an open dataset of 14,566 CUDA tasks from 41 sources, paired with verified solutions and trajectories (Figure 1). Human-guided review improves specification quality, while diversity-aware selection broadens computational coverage. The KernelMosaic Framework then constructs the solutions and trajectories by guiding a strong LLM through implementation, correctness repair, and performance optimization. Reusable CUDA

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

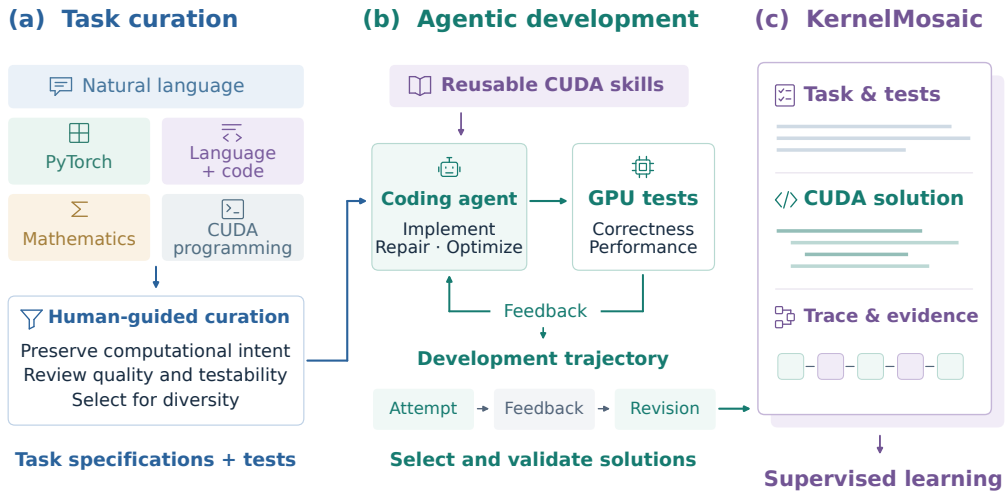


Figure 1: Construction of KernelMosaic. **(a)** Human-guided curation preserves computational intent, reviews quality and testability, and selects for diversity across specification forms. **(b)** The agentic framework equips a coding agent with reusable CUDA skills and correctness and performance feedback to develop implementations and record attempts and observations. **(c)** The dataset links tasks and tests with selected CUDA solutions, trajectories, and evaluation evidence for supervised learning.

guidance and execution feedback support this iterative development, and the recorded interactions provide supervision for training kernel-generation models.

We demonstrate the dataset's utility by fine-tuning Qwen3.5-9B (Qwen Team, 2026a). On KernelBench (Ouyang et al., 2025) L1, L2, and L3, correctness increases from 4%, 0%, and 0% to 88%, 82%, and 16%, respectively, with more tasks yielding kernels faster than PyTorch eager at every level. In a separate comparison using 1,000 examples and one training epoch, trajectory supervision yields 31 correct solutions, compared with one for direct-response supervision. Trajectory-trained models also engage in more sustained testing and revision, connecting the construction of training data with improved kernel-development behavior.

Our contributions are threefold. **First,** we introduce KernelMosaic, an open dataset combining diverse task specifications with verified CUDA solutions, trajectories, tests, and evaluation evidence. **Second,** we develop an agentic framework for constructing and extending this dataset through correctness repair and performance optimization. **Third,** we establish the dataset's utility for supervised learning through substantial benchmark gains and analyses of how trajectory supervision improves kernel-generation behavior.

2 Related Work

Kernel datasets and learning. GPU programming datasets span several supervision formats (Table 1). LS-CAT pairs CUDA kernels with autotuning measurements (Bjertnes et al., 2021), while CASS aligns source and assembly across NVIDIA and AMD platforms (Heakl et al., 2026). KernelBook supplies paired PyTorch and Triton programs (Paliskara & Saroufim, 2025); AutoTriton and ConCuR construct code-and-reasoning supervision for Triton and CUDA, respectively (Li et al., 2025b; Kong et al., 2025). CUDA Agent combines synthesized tasks with an agentic reinforcement-learning environment (Dai et al., 2026). Kevin learns through multi-turn reinforcement learning (Baronio et al., 2026), CUDA-L1 through contrastive optimization (Li et al., 2026), and SparseRL targets sparse-matrix kernels (Wang et al., 2026). KernelMosaic brings heterogeneous specifications together with solutions and revision trajectories for supervised learning.

Agentic kernel generation. CUDA-LLM uses feature search and reinforcement to refine CUDA programs (Chen et al., 2025). CudaForge combines coding and judging agents with hardware feedback (Zhang et al., 2025b), while CUDA-Harness couples natural-language specifications with synthesized verification and iterative refinement (Fan et al., 2026). Atrex-Kernel-Agent uses a GPU knowledge base (Yang et al., 2026); ReGraphT transfers optimization experience to smaller models through retrieval and search over reasoning graphs (Gong et al., 2026). Dr. Kernel introduces KernelGYM for distributed evaluation, multi-turn trajectory collection, and reinforcement learning (Liu et al., 2026). AutoGuide structures execution feedback for parallel-program mapping optimization (Wei et al., 2025). Our framework organizes tools, execution feedback, and domain guidance around dataset construction across multiple task representations.

Evaluation and task coverage. KernelBench measures functional correctness and speed across PyTorch workloads of increasing complexity (Ouyang et al., 2025). TritonBench evaluates Triton generation on real-world operators and tasks aligned with PyTorch interfaces (Li et al., 2025a). Atrex-Bench derives workloads from production traces (Yang et al., 2026). We curate training coverage through source-aware review and semantic selection using Qwen3 embeddings (Zhang et al., 2025a).

3 Task collection and curation

Task curation determines which computational requirements the dataset represents and whether those requirements can support meaningful implementation and evaluation. Our method combines source-aware construction, human-guided quality assessment, diversity selection, and preparation of executable tests.

3.1 Preserving heterogeneous specifications

We collect reference programs, existing instructions, paired implementations, and mathematical specifications. Each task describes a target computation, the requested CUDA implementation, and its source. Five categories capture the principal input forms and programming settings: natural language, PyTorch code, mixed language and code, algebraic or matrix specifications, and CUDA programming tasks such as source completion.

Construction follows the information supplied by each source. Translation tasks expose the reference program and withhold the target implementation. Completion tasks preserve the surrounding code needed to infer missing behavior. Mathematical tasks specify dimensions, domains, numerical types, and outputs while leaving implementation choices open. Across these forms, interface conventions and boundary conditions remain part of the specification. This treatment preserves meaningful differences between tasks while giving each a self-contained implementation objective.

3.2 Human-guided quality assessment

Human inspection and AI-assisted review identify ambiguous requirements, missing context, unsuitable computational targets, and unnecessary solution guidance. Review findings refine source-specific construction rules, allowing a correction to improve other examples with the same underlying issue. Known semantic defects and non-CUDA targets are excluded. For tasks not reviewed individually, admission relies on source-level audits or upstream validation with verified input extraction. Appendix D specifies these admission rules.

Review also distinguishes substantive variation in specifications from differences in presentation. Mixed language-and-code tasks combine program context with requirements expressed in prose. Review corrects category labels, repairs incomplete specifications, and excludes unsuitable tasks while retaining differences in computational requirements and input form.

3.3 Diversity-aware selection

We augment underrepresented mathematical inputs with source-grounded and synthetic specifications, and remove normalized duplicates. We then use Qwen3-Embedding-8B (Zhang et al., 2025a)

for weighted farthest-first selection over the combined pool. Each task receives a weight w_i combining source provenance, recorded review scores, input category, mathematical reasoning, and CUDA relevance.

For normalized task embeddings z_i and retained set S, the similarity of a task to its closest retained representative is

$$\text{coverage}(i, S) = \max_{j \in S} z_i^\top z_j. \tag{1}$$

Starting with the highest-weight task, selection repeatedly adds the uncovered task maximizing $w_i[1 - \text{coverage}(i, S)]$. A task is covered when its similarity to a retained representative reaches the cutoff. This direct comparison prevents a chain of moderately similar tasks from obscuring an uncovered example. Weights, thresholds, and the additional mathematical-task selection are given in Appendix D.

3.4 Preparing executable evaluation

We recover upstream references and tests, and supplement them with generated test cases and implementation interfaces. Evaluation resources specify dependencies, expected outputs, and numerical tolerances. Task clarity and evaluability are assessed separately because a well-defined computation may still require additional inputs or fixtures to execute.

We assess test suites through reference execution and candidate controls, using valid implementations and deliberately corrupted outputs to probe checker behavior. Evaluation records distinguish candidate tests, reference-only checks, and restricted-buffer comparisons. Compilation, reference execution, and candidate correctness are recorded separately. Tasks without a usable evaluation procedure are excluded from trajectory generation. These resources provide feedback for implementation development.

4 Agentic kernel and trajectory generation

The framework builds the dataset's solutions and development trajectories. A model-facing harness, task-specific evaluation, and a revision controller jointly develop and select implementations across specification forms and model configurations.

4.1 Reusable CUDA guidance

The harness builds on OpenCode (OpenCode Contributors, 2026), providing tools for code editing, execution, and inspection. Reusable CUDA skills organize guidance on numerical behavior, memory access, parallel computation, launch overhead, fusion, and performance diagnosis. The harness exposes this guidance through callable skills that connect optimization knowledge to task-specific implementation decisions.

Skills support direct invocation and optional routing. Routing associates guidance with development stages and observed failures, such as numerical mismatches or repeated compilation errors, and suggests relevant skills for the next revision.

4.2 Implementation, repair, and optimization

Given task x_i and development history h_t, the agent proposes a candidate implementation c_t. Evaluation returns compilation outcomes, numerical checks, and performance measurements. This feedback guides repair until a candidate passes the correctness tests, after which further revisions seek improved performance. Previously valid implementations remain available throughout optimization, and resource constraints bound the development process.

Let $\mathcal{C}_i$ denote the candidate set, $v_i^{\text{pub}}(c)$ indicate correctness on tests visible during development, and $s_i^{\text{pub}}(c)$ denote the corresponding performance score. The controller selects

$$c_i^\star = \arg \max_{c \in \mathcal{C}_i :\, v_i^{\text{pub}}(c)=1} s_i^{\text{pub}}(c). \tag{2}$$

Table 1: Related datasets for GPU kernel learning. Scale uses each resource's native unit. Each of the 14,566 tasks in KernelMosaic has an associated development trajectory. Mixed denotes multiple source and assembly languages.

Dataset	Domain	Scale	Data / supervision
LS-CAT (Bjertnes et al., 2021)	CUDA	19,683 kernels	Code and runtimes
KernelBook (Paliskara & Saroufim, 2025)	Triton	18,162 pairs	PyTorch–Triton pairs
AutoTriton SFT (Li et al., 2025b)	Triton	14,102 examples	Code and reasoning
ConCuR (Kong et al., 2025)	CUDA	4,892 examples	Code and reasoning
CASS (Heakl et al., 2026)	Mixed	60,694 samples	Cross-platform pairs
CUDA-Agent-Ops-6K (Dai et al., 2026)	CUDA	6,000 tasks	PyTorch tasks, tests
KernelMosaic	CUDA	14,566 tasks	Solutions, trajectories

Tasks without a passing candidate retain a failure outcome. Held-out tests evaluate the selected implementation only after development; their inputs are never exposed to the agent.

4.3 Qualification and trajectory construction

Solutions are admitted by the correctness checks specified in each task's evaluation policy. The policy determines the use of public and held-out tests and defines requirements on output structure, input mutation, and numerical tolerances. Performance is measured against a specified reference on a defined workload and device. Correctness and speed are recorded separately.

The framework records model responses, candidate programs, revisions, and evaluation feedback alongside the selected implementation. The resulting trajectory captures successful changes as well as failed attempts that informed later decisions. Task identity and evaluation outcomes connect this history to the computation it addresses. These associations support learning from both final solutions and the process of developing them.

The dataset supports solution-based and trajectory-based training views. Our supervised study uses trajectories segmented into complete assistant decisions and associated tool observations, as detailed in Appendix A.1. The same framework supports additional data generation and kernel development with other models. Implementation and optional explanation-generation details appear in Appendix D.

5 The KernelMosaic dataset

5.1 Tasks, solutions, and development trajectories

KernelMosaic associates each task with its specification, source, available tests, generated implementations, and trajectory. Evaluation outcomes identify the selected solution and its assessment conditions. Tasks support fresh generation experiments, solutions provide implementation supervision, and trajectories expose repair and optimization behavior. Failed intermediate attempts preserve the feedback that motivated subsequent revisions.

5.2 Dataset composition

The dataset contains 14,566 tasks from 41 sources, each paired with a successful trajectory generated by Qwen3.8-Flash (Qwen Team, 2026b) under fixed token and time budgets and an implementation that passes the task's evaluation checks. Table 1 compares resources for CUDA, Triton, and cross-platform GPU programming. Appendix D gives category counts and generation budgets.

The dataset comprises 13,815 training and 751 validation task–trajectory records across all five input categories. PyTorch programs account for 68.4% of tasks, alongside CUDA programming, mixed language-and-code, algebraic, and natural-language specifications. Appendix E examines computational vocabulary, category coverage, source profiles, and supervision.

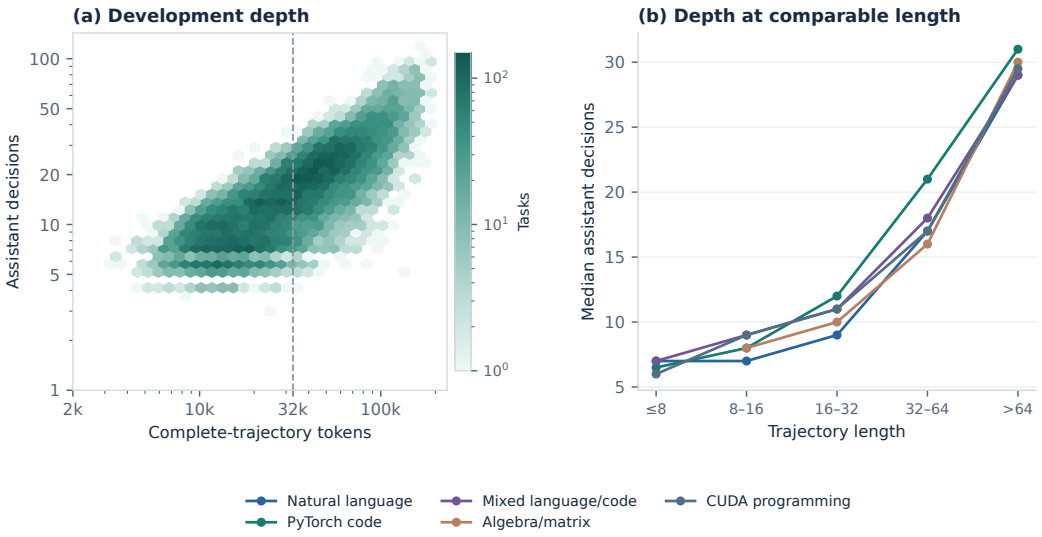


Figure 2: Length and interaction depth of the 14,566 trajectories. **(a)** Density of complete-trajectory tokens and assistant decisions on logarithmic scales; the dashed line marks the 32,768-token training context. **(b)** Median decision counts by input category within token-length intervals ($\times$1,024). Intervals are right-closed; points with fewer than 20 tasks are omitted. Length includes the complete context and historical reasoning.

5.3 Learning from development trajectories

Trajectories contain a median of 14 assistant decisions, and 47.4% exceed the 32,768-token training context limit (Figure 2a). Interaction depth also varies at comparable token lengths: within 32,768–65,536 tokens, median decision counts range from 16 for algebraic tasks to 21 for PyTorch tasks; above 65,536 tokens, all five input categories have medians between 29 and 31 (Figure 2b). These trajectories provide supervision across multiple rounds of implementation and revision.

We divide long trajectories at complete assistant-decision and tool-interaction boundaries, producing 37,242 training and 2,010 validation windows. Each window preserves the task and relevant history; only newly assigned assistant decisions contribute to the loss. Repeated history and tool observations provide context without duplicating supervision. Appendix A specifies window construction, and Appendix G illustrates tasks and a measured LayerNorm revision.

6 Experiments

We evaluate the utility of KernelMosaic for learning CUDA kernel generation. KernelBench covers individual operations, operation sequences, and model-level computations; we report correctness and performance at each level (Ouyang et al., 2025).

6.1 Experimental setup

We compare Qwen3.5-9B (Qwen Team, 2026a) with full-parameter SFT on KernelMosaic for three epochs over 37,242 training windows. Two one-epoch variants with 1,000 examples each compare trajectory and direct-response supervision. All use a 32,768-token context limit and AdamW with a constant learning rate of 10^{-5}. Appendix A provides the training configurations and supervision objective.

All four checkpoints use a common agentic evaluation protocol on KernelBench v0.1's 100 L1, 100 L2, and 50 L3 tasks. Each task requests CUDA C++ code through the reference module's Python interface and permits one trajectory with up to 262,144 generated tokens and 3,600 active seconds,

Table 2: KernelBench correctness (C) and faster-than-PyTorch-eager coverage (fast_1), in percent. The 1k variants each use 1,000 training examples and one epoch. Published results use different evaluation protocols. Rows from Li et al. (2025b) generate Triton; the others generate CUDA. – denotes an unreported result. Appendix C gives protocol details and additional speedup thresholds.

Method / model	Inference	L1		L2		L3	
		C	fast_1	C	fast_1	C	fast_1
Open-weight models							
Qwen3-8B (Li et al., 2025b)	Single-turn	16	9	16	8	14	4
DeepSeek-R1-0528 (Li et al., 2025b)	Single-turn	35	7	42	28	26	14
Kernel-specialized open-weight models							
AutoTriton-8B (Li et al., 2025b)	Single-turn	36	10	45	17	20	10
Kevin-32B (Kong et al., 2025)	Single-turn	50	16	46	27	–	–
KernelCoder-32B (Kong et al., 2025)	Single-turn	58	17	59	39	–	–
Proprietary models and systems							
GPT-4o (Li et al., 2025b)	Single-turn	15	3	5	3	8	4
Claude Sonnet 4 (Li et al., 2025b)	Single-turn	33	11	26	10	18	2
Claude Opus 4.5 (Dai et al., 2026)	Agentic	96	88	98	97	88	82
Gemini 3 Pro (Dai et al., 2026)	Agentic	95	90	93	91	80	76
CudaForge (o3) (Zhang et al., 2025b)	Multi-turn	96	54	100	89	96	68
CUDA Agent (Seed1.6) (Dai et al., 2026)	Agentic	100	99	100	100	94	94
Our Qwen3.5-9B comparison							
Qwen3.5-9B	Agentic	4	0	0	0	0	0
+ Single-turn SFT (1k)	Agentic	1	1	0	0	0	0
+ Trajectory SFT (1k)	Agentic	23	13	8	4	0	0
+ Trajectory SFT (full)	Agentic	88	64	82	62	16	12

excluding inference and infrastructure waits. Inference runs on B200 GPUs; candidate evaluation uses H200 GPUs.

A task is solved when an accepted candidate passes the numerical checks; its performance is the largest retained speedup among these candidates. Timing compares synchronized forward-call wall time against PyTorch eager, including Python dispatch. Appendix B specifies correctness checks, timing, and budget accounting.

Metrics. For N tasks, let v_i indicate whether a task is solved and s_i denote its best retained reference-to-candidate speedup. Following KernelBench, we report correctness and

$$\text{fast}_p = \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}\{v_i = 1 \ \wedge \ s_i > p\}, \tag{3}$$

for $p \in \{1, 2\}$. Each metric includes all tasks in the denominator, including unsuccessful and timed-out attempts. A correct candidate without a reliable timing measurement contributes to correctness but not to fast_p.

6.2 Kernel-generation performance

Table 2 reports results by level alongside published models and systems. The comparison covers general and kernel-specialized open-weight models, frontier proprietary models, and iterative systems. Within our Qwen3.5-9B comparison, fine-tuning increases L1 correctness from 4% to 88% and fast_1 from 0% to 64%. On L2, the original model produces no correct candidates, whereas the fine-tuned model achieves 82% correctness and 62% fast_1. On L3, correctness increases from 0% to 16% and fast_1 from 0% to 12%.

Figure 3 shows performance across speedup thresholds. Fine-tuning yields fast_2 of 31%, 51%, and 6% on L1, L2, and L3, respectively.

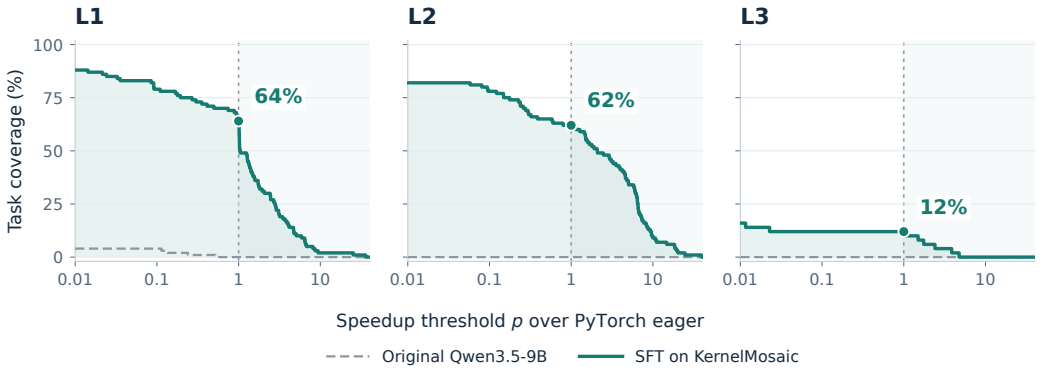


Figure 3: Correctness and speed of the original and full-dataset SFT checkpoints on KernelBench. Separately, L1, L2, and L3 contain 100 individual operators, 100 operator sequences, and 50 model-level workloads. Curves show the fraction of tasks whose best correct candidate exceeds speedup p over PyTorch eager. Vertical guides mark equal latency; labeled points give the SFT model's fast_1 scores. Each model has one trajectory per task, evaluated on H200.

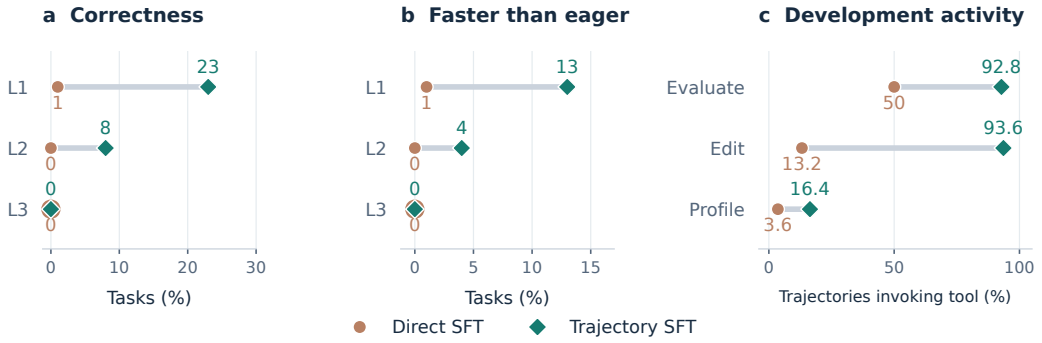


Figure 4: Supervision from agentic trajectories and direct responses. Both variants train Qwen3.5-9B for one epoch on 1,000 independently sampled examples. **(a–b)** KernelBench correctness and faster-than-eager coverage by level. **(c)** Fraction of the 250 evaluation trajectories invoking each tool. Trajectory supervision improves L1/L2 performance and supports more frequent evaluation and editing.

6.3 Effect of supervision construction

Trajectory SFT uses 1,000 successful framework-generated trajectories containing reasoning, tool use, and revisions conditioned on execution feedback. **Direct SFT** uses 1,000 one-shot reasoning-and-code responses generated without the framework or correctness filtering. The cohorts are independently sampled; both models start from Qwen3.5-9B and train for one epoch with the same learning rate, global batch size, and context limit. Appendix A.3 details cohort selection, token counts, and optimization.

Across all 250 tasks, Trajectory SFT yields 31 correct solutions and 17 faster-than-eager solutions, versus one of each for Direct SFT (Table 2; Figure 4a–b). The gains span L1 and L2; neither variant solves an L3 task. Evaluated agentic iteration thus produces more effective training data than direct responses in this comparison.

Trajectory supervision also promotes sustained testing and revision (Figure 4c): evaluation occurs on 92.8% of tasks and editing on 93.6%, compared with 50.0% and 13.2% for Direct SFT. Median tool calls increase from 4 to 105. Appendix F.5 analyzes this activity by level.



Figure 5: Original and full-dataset SFT behavior on KernelBench. **(a)** Mean within-trajectory tool-use shares; Write creates or replaces files, while Edit modifies existing contents. **(b)** Correct (solid) and faster-than-eager (dashed) task coverage versus cumulative output tokens ($\times$1,024). Each panel includes all tasks at its level.

6.4 Model behavior during kernel development

Comparing the original checkpoint with full-dataset SFT across 500 benchmark trajectories, SFT shifts tool use from file writes toward editing, code inspection, and evaluation (Figure 5a). Repeated file contents account for 60.1% of the original model's completed writes, compared with 2.6% after SFT. Profiling appears in 87%, 85%, and 30% of SFT trajectories on L1–L3, versus 6%, 4%, and 2% for the original model.

By 65,536 output tokens, SFT solves 66% of L1 and 59% of L2 tasks, compared with 1% and 0% for the original model (Figure 5b). L3 correctness reaches 8% at this point and 16% at the full budget.

Development continues after correctness. SFT's first timed correct candidates outperform PyTorch eager on 49% of L1 and 46% of L2 tasks. Subsequent distinct implementations raise this coverage to 64% and 62%, respectively. On L3, six of the eight solved tasks exceed eager performance at their first timed correct candidate. Appendix F analyzes action sequences, recovery, resource use, and task-level optimization.

7 Conclusion

KernelMosaic links diverse computational specifications with tested CUDA solutions and agentic trajectories. Its framework builds this supervision through iterative implementation, repair, and optimization. Fine-tuning Qwen3.5-9B raises KernelBench L1, L2, and L3 correctness from 4%, 0%, and 0% to 88%, 82%, and 16%, respectively, and faster-than-eager coverage from zero to 64%, 62%, and 12%. With 1,000 examples and one epoch, training on framework-generated trajectories solves 31 benchmark tasks, compared with one for direct-response supervision. The learned behavior emphasizes evaluation and targeted editing, connecting the framework's data-construction process with effective iterative development. KernelMosaic enables a general-purpose model to achieve large gains in both correct and faster-than-eager CUDA generation through iterative agentic workflows.

AI ASSISTANCE STATEMENT

AI assistants supported task and test construction, code implementation, literature retrieval, and manuscript drafting and editing. Human inspection and automated evaluation informed task curation and implementation assessment.

REPRODUCIBILITY STATEMENT

Appendix A specifies the training representation, data splits, objective, and optimization settings. Appendix B describes benchmark adaptation, correctness tests, timing, and resource constraints. Additional appendices provide curation parameters, trajectory analyses, and illustrative tasks. The underlying records associate each task with its source, generated implementations, development history, and evaluation outcomes. We release the dataset and generation framework.

ETHICS STATEMENT

The dataset incorporates third-party programs and model-assisted material. The release retains source attribution and upstream licensing information in its metadata. Generated GPU programs should run in controlled environments with resource limits.

REFERENCES

Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. Kevin: Multi-turn RL for generating CUDA kernels. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/86ba836d4c5dd859d795a172911745e2-Abstract-Conference.html.

Lars Bjertnes, Jacob O. Tørring, and Anne C. Elster. LS-CAT: A large-scale CUDA autotuning dataset. In *2021 IEEE International Conference on Applied Artificial Intelligence (ICAPAI)*. IEEE, 2021. doi: 10.1109/ICAPAI49758.2021.9462050. URL https://ieeexplore.ieee.org/document/9462050.

Wentao Chen, Jiace Zhu, Qi Fan, Yehan Ma, and An Zou. CUDA-LLM: Llms can write efficient cuda kernels. *arXiv preprint arXiv:2506.09092*, 2025. URL https://arxiv.org/abs/2506.09092.

Weinan Dai, Hanlin Wu, Qiying Yu, Huan-ang Gao, Jiahao Li, Chengquan Jiang, Weiqiang Lou, Yufan Song, Hongli Yu, Jiaze Chen, Wei-Ying Ma, Ya-Qin Zhang, Jingjing Liu, Mingxuan Wang, Xin Liu, and Hao Zhou. CUDA Agent: Large-scale agentic rl for high-performance cuda kernel generation. *arXiv preprint arXiv:2602.24286*, 2026. URL https://arxiv.org/abs/2602.24286.

Qi Fan, An Zou, and Yehan Ma. CUDA-Harness: Harnessing agentic cuda kernel generation and optimization from natural language. *arXiv preprint arXiv:2609.00058*, 2026. URL https://arxiv.org/abs/2609.00058.

Fhrozen. Stack-Prompts. Dataset, CUDA subset, 2026. URL https://huggingface.co/datasets/Fhrozen/stack-prompts. Accessed September 26, 2026.

Junfeng Gong, Zhiyi Wei, Junying Chen, Cheng Liu, and Huawei Li. From large to small: Transferring CUDA optimization expertise via reasoning graph. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/c4e40d310af1be55f2c0f32cb16a9a16-Abstract-Conference.html.

Ahmed Heakl, Gustavo Bertolo Stahl, Sarim Hashmi, Seung Hun Eddie Han, Mukul Ranjan, Arina Kharlamova, Salman Khan, and Abdulrahman Mahmoud. CASS: Nvidia to AMD transpilation with data, models, and benchmark. In *Proceedings of the 64th Annual Meeting of the*

Association for Computational Linguistics (Volume 1: Long Papers), pp. 34489–34508. Association for Computational Linguistics, 2026. URL https://aclanthology.org/2026.acl-long.1592/.

Lingcheng Kong, Jiateng Wei, Hanzhang Shen, and Huan Wang. ConCuR: Conciseness makes state-of-the-art kernel generation. *arXiv preprint arXiv:2510.07356*, 2025. URL https://arxiv.org/abs/2510.07356.

Jianling Li, Shangzhan Li, Zhenye Gao, Qi Shi, Yuxuan Li, Zefan Wang, Jiacheng Huang, Haojie Wang, Jianrong Wang, Xu Han, Zhiyuan Liu, and Maosong Sun. TritonBench: Benchmarking large language model capabilities for generating Triton operators. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 23053–23066. Association for Computational Linguistics, 2025a. doi: 10.18653/v1/2025.findings-acl.1183. URL https://aclanthology.org/2025.findings-acl.1183/.

Shangzhan Li, Zefan Wang, Ye He, Yuxuan Li, Qi Shi, Jianling Li, Yonggang Hu, Wanxiang Che, Xu Han, Zhiyuan Liu, and Maosong Sun. AutoTriton: Automatic triton programming with reinforcement learning in llms. *arXiv preprint arXiv:2507.05687*, 2025b. URL https://arxiv.org/abs/2507.05687.

Xiaoya Li, Xiaofei Sun, Albert Wang, Jiwei Li, and Chris Shum. CUDA-L1: Improving CUDA optimization via contrastive reinforcement learning. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/c94bbbef466ab1b2cfa100e41413b3a8-Abstract-Conference.html.

Wei Liu, Jiawei Xu, Yingru Li, Longtao Zheng, Tianjian Li, Qian Liu, and Junxian He. Dr. Kernel: Reinforcement learning done right for triton kernel generations. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026. URL https://arxiv.org/abs/2602.05885.

llama.cpp Contributors. llama.cpp. Software, 2026. URL https://github.com/ggml-org/llama.cpp. Accessed September 26, 2026.

NVIDIA. Nemotron-SFT-CUDA-v1. Dataset, 2026. URL https://huggingface.co/datasets/nvidia/Nemotron-SFT-CUDA-v1. Accessed September 26, 2026.

OpenCode Contributors. OpenCode, 2026. URL https://github.com/anomalyco/opencode. Upstream coding-agent infrastructure.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 47356–47415. PMLR, 2025. URL https://proceedings.mlr.press/v267/ouyang25a.html.

Sahan Paliskara and Mark Saroufim. KernelBook. Dataset, 2025. URL https://huggingface.co/datasets/GPUMODE/KernelBook.

Qwen Team. Qwen3.5-9B model card, 2026a. URL https://huggingface.co/Qwen/Qwen3.5-9B. Accessed September 20, 2026.

Qwen Team. Qwen3.8-Flash. QwenCloud model documentation, 2026b. URL https://docs.qwencloud.com/developer-guides/getting-started/latest-model. Accessed September 26, 2026.

Yaoyu Wang, Hankun Dai, Zhidong Yang, Junmin Xiao, and Guangming Tan. Mastering sparse CUDA generation through pretrained models and deep reinforcement learning. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/32400dbdd8b99abe47a49e43dcb1fcf5-Abstract-Conference.html.

Anjiang Wei, Allen Nie, Thiago S. F. X. Teixeira, Rohan Yadav, Wonchan Lee, Ke Wang, and Alex Aiken. Improving parallel program performance with LLM optimizers via agent-system interfaces. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 66155–66177. PMLR, 2025. URL https://proceedings.mlr.press/v267/wei25j.html.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. In *Advances in Neural Information Processing Systems*, volume 38, 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/8b86cf5ace600c48fd188efbb8dedec8-Abstract-Datasets_and_Benchmarks_Track.html.

Lingyun Yang, Yuxiao Wang, Shenghao Liang, Linfeng Yang, Daocheng Ying, Chunbo You, Rui Zhang, Luping Wang, Yinghao Yu, Guodong Yang, and Liping Zhang. Are llm-generated gpu kernels production-ready? a trace-driven benchmark and optimization agent. *arXiv preprint arXiv:2607.14541*, 2026. URL https://arxiv.org/abs/2607.14541.

Z.ai. GLM-5.3 model card, 2026. URL https://huggingface.co/zai-org/GLM-5.3. Accessed September 26, 2026.

Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, Fei Huang, and Jingren Zhou. Qwen3 Embedding: Advancing text embedding and reranking through foundation models. *arXiv preprint arXiv:2506.05176*, 2025a. URL https://arxiv.org/abs/2506.05176.

Zijian Zhang, Rong Wang, Shiyang Li, Yuebo Luo, Mingyi Hong, and Caiwen Ding. CudaForge: An agent framework with hardware feedback for cuda kernel optimization. *arXiv preprint arXiv:2511.01884*, 2025b. URL https://arxiv.org/abs/2511.01884.

A SUPERVISED TRAINING DETAILS

We train Qwen3.5-9B by full-parameter supervised fine-tuning on development trajectories from KernelMosaic. The initialization is the checkpoint used in the original-model comparison.

A.1 TRAJECTORY REPRESENTATION AND SUPERVISION

Windows are constructed independently within the task-disjoint training and validation splits summarized in Table 3. Training covers all 41 sources and five input categories.

To accommodate long trajectories, we divide them into windows at complete decision boundaries. A decision comprises an assistant message and its associated tool observations, or an assistant message without a tool call. Each window retains the original task and all user constraints encountered so far. Historical assistant reasoning is removed while prior actions and tool observations remain as context. Complete target decisions are added within the 32,768-token limit. If the next target cannot fit, the oldest complete historical decisions are removed as needed. A target that still cannot fit with the required task context is excluded as a whole.

Each retained decision contributes supervision exactly once across windows. Newly assigned assistant reasoning, tool calls, and text are supervised; task context, tool observations, and repeated historical assistant content are masked. For window z_i and binary target mask m_{it}, the objective is

$$\mathcal{L}(\theta) = -\frac{\sum_{i,t} m_{it} \log p_\theta(z_{it} \mid z_{i,<t})}{\sum_{i,t} m_{it}}. \tag{4}$$

Each training example contains one complete exported window. Padding positions are excluded from the loss.

This procedure retains 252,371 of 253,044 decisions (99.7%) and excludes 673 oversized decisions. Every source trajectory contributes at least one example. Of 39,252 windows, 10,796 (27.5%) require removal of some historical context. Table 3 reports the resulting trajectory, window, and token counts per copy of each split.

Table 3: Trajectory and token counts for supervised learning. Total tokens include context; supervised tokens contribute to the loss.

Split	Trajectories	Windows	Total tokens	Supervised tokens
Training	13,815	37,242	1,002,791,260	416,424,437
Validation	751	2,010	54,137,600	22,460,320
Total	14,566	39,252	1,056,928,860	438,884,757

A.2 OPTIMIZATION

Table 4 specifies the optimization settings. We evaluate the final checkpoint after three epochs; Appendix B gives the inference and evaluation protocol.

Table 4: Full-dataset supervised fine-tuning configuration.

Parameter	Value
Model / trainable parameters	Qwen3.5-9B / all 8,953,803,264
Optimizer steps	13,968
Per-device batch / gradient accumulation	1 / 1
Global batch / training GPUs	8 / eight B200 GPUs
Optimizer / learning rate / warm-up	AdamW / constant 10^{-5} / none
Gradient clipping / model and data seeds	1.0 / 42
Sequence length / packing	32,768 / none
Precision / sharding / attention	BF16 / FSDP2 / FlashAttention-2
Activation checkpointing	26 of 32 layers

A.3 TRAJECTORY AND DIRECT-RESPONSE SUPERVISION

The 1,000-example comparison evaluates two supervision-construction approaches, both using Qwen3.8-Flash to generate the training responses. For Trajectory SFT, we uniformly sample 1,000 trajectories from the 13,815-task training split and retain every associated decision window. The sample contains 2,519 windows, with 17,050 supervised decisions; 51 oversized decisions are excluded by the window-construction rule above. Direct SFT uses one-shot reasoning-and-code responses to curated task specifications, without tools, execution feedback, or correctness filtering. We preserve each complete system–user–assistant sequence and sample 1,000 of the 1,079 responses that fit within 32,768 tokens, from a source collection of 5,164 responses. Sampling uses seed 20260923 in both cases. The independently selected cohorts share 61 task IDs.

Both variants use the original Qwen3.5-9B initialization, full-parameter AdamW optimization, a constant learning rate of 10^{-5}, global batch size eight, BF16, FSDP2, FlashAttention-2, and model/data seed 42. They train for one epoch with no packing or learning-rate warm-up. Trajectory targets follow the decision-level masking described above; Direct SFT supervises the complete assistant reasoning and answer while masking the system and user messages. Table 5 reports the resulting training exposure. Trajectory SFT runs on four B200 GPUs with gradient accumulation two; Direct SFT runs on eight H200 GPUs with accumulation one.

Table 5: Training data and optimization for the one-epoch supervision comparison. Tokens are counted before repetition across epochs; total tokens include context.

	Direct SFT	Trajectory SFT
Source examples	1,000	1,000
Sources	20	23
Natural-language tasks	41	32
PyTorch specifications	31	677
Mixed language and code	264	117
Algebraic or matrix specifications	12	30
CUDA programming tasks	652	144
Training windows	1,000	2,519
Total tokens	12,594,890	66,957,025
Supervised tokens	11,416,743	28,947,222
Optimizer steps	125	315

B KERNELBENCH EVALUATION PROTOCOL

We evaluate all four checkpoints on all 250 tasks in KernelBench v0.1: 100 L1, 100 L2, and 50 L3 tasks. Each model generates one trajectory per task.

B.1 TASK ADAPTATION AND INFERENCE

The adapter preserves the upstream reference programs and input generators. It requests a CUDA C++ implementation through a Python module that maintains the reference constructor and forward interface. Tools support file reading, writing, editing, GPU evaluation, and profiling. Thinking and native context compaction are enabled with the default Qwen chat template. Each response permits 32,768 tokens, including a reasoning budget of 24,576 tokens. Task-level limits are 262,144 generated tokens and 3,600 active seconds, including a 300-second finalization reserve. The active clock excludes model inference and infrastructure waits; generated reasoning and response tokens count toward the token limit.

Checkpoints are served in pairs, each through four single-GPU replicas on B200 devices. The original and full-dataset SFT pair precedes the two 1,000-example variants. Compilation precedes GPU evaluation, which assigns each candidate exclusive access to one of eight H200 NVL devices.

B.2 CORRECTNESS AND TIMING

A candidate must execute CUDA code and match the reference outputs on three input seeds (104729, 130363, 155921), with absolute and relative tolerances both 0.01. The evaluation tools return numerical and performance feedback during the trajectory.

Measurements use PyTorch 2.7.1 with CUDA 12.6. Before formal evaluation, we warm up the GPU until its operating frequency stabilizes around 1,785 MHz. Forward-call timings synchronize the GPU and include Python dispatch. Constructors, input cloning, compilation, and numerical comparison are outside the timed interval. For each implementation, we additionally perform one warm-up pass and three measured passes over the three input seeds. The timing stage therefore executes twelve forward calls per implementation: three warm-up calls and nine measured calls. Numerical checks are repeated during measurement. Speedup is the median of the nine reference latencies divided by the median of the nine candidate latencies. Timing measurements are retained when the relative median absolute deviation is at most 25% for both the reference and candidate.

B.3 TRAJECTORY SCORING

Scoring pools the candidate evaluations produced within both task-level resource limits. A task contributes to correctness when at least one candidate passes the numerical and CUDA execution checks. Its performance score is the largest retained speedup among these correct candidates. The active-time and cumulative output-token counts recorded at each evaluation determine eligibility. For interrupted responses with missing token usage, the response token limit provides an upper bound before budget eligibility is determined. Every scheduled task contributes to the denominator for correctness and fast_p.

All 178 tasks solved by the fine-tuned model and all four solved by the original checkpoint have retained timing measurements.

The one-epoch comparison retains three additional passing candidates whose recorded usage is below the limits but whose output-token upper bounds are unresolved because some responses lack usage metadata: two Trajectory SFT tasks on L2 and one Direct SFT task on L1. Their correctness checks and timing measurements are retained in the reported scores. Restricting scores to candidates with verified budget bounds gives Trajectory SFT correctness of 23%, 6%, and 0% and fast_1 of 13%, 3%, and 0% on L1–L3; Direct SFT scores are zero at all levels. All 31 Trajectory SFT and one Direct SFT successes have retained timing measurements.

C PUBLISHED KERNELBENCH COMPARISONS

Table 2 places our learning study in the context of published KernelBench results. All rows retain the original authors' measurements. We summarize each source's generation language, hardware, inference budget, and candidate-selection procedure below.

The inference labels distinguish single-completion generation, a fixed multi-turn refinement workflow, and an agentic tool-use loop. CudaForge uses coding and judging agents within prescribed refinement rounds and is labeled multi-turn. PyTorch-reference results use the eager baseline unless a compiled mode is specified.

Direct generation. Qwen3-8B, DeepSeek-R1-0528, AutoTriton-8B, GPT-4o, and Claude Sonnet 4 values come from Table 2 of Li et al. (2025b), using its Triton adaptation of KernelBench. We reproduce execution accuracy and fast_1. The paper does not specify the evaluation GPU or exact sampling budget. Kevin-32B and KernelCoder-32B values are the pass@1 results in Table 1 of Kong et al. (2025), evaluated on RTX 5090 GPUs against PyTorch eager. L3 is not reported there. Kevin was trained on 180 of the 200 L1/L2 tasks (Baronio et al., 2026).

Iterative systems. CudaForge values are from Table 2 of Zhang et al. (2025b), covering the full benchmark on RTX 6000 GPUs with OpenAI-o3 as both coder and judge. The default budget is ten rounds, and the best correct candidate is selected. Claude Opus 4.5, Gemini 3 Pro, and CUDA Agent values are from Table 1 of Dai et al. (2026), using the eager-reference faster-rate column. These models use the same agent loop, allowing up to 200 evaluation turns, a 131,072-token context,

Table 6: KernelBench speedup coverage (%), with F_p denoting fast$_p$ relative to PyTorch eager. Published results use different evaluation protocols. The published rows generate Triton in a single turn; our rows generate CUDA with agentic inference.

Model	L1		L2		L3	
	F_1	F_2	F_1	F_2	F_1	F_2
Qwen3-8B (Li et al., 2025b)	9	9	8	1	4	2
DeepSeek-R1-0528 (Li et al., 2025b)	7	1	28	2	14	2
AutoTriton-8B (Li et al., 2025b)	10	6	17	0	10	4
GPT-4o (Li et al., 2025b)	3	1	3	0	4	2
Claude Sonnet 4 (Li et al., 2025b)	11	7	10	1	2	0
Our Qwen3.5-9B comparison						
Qwen3.5-9B	0	0	0	0	0	0
+ Direct SFT (1k)	1	1	0	0	0	0
+ Trajectory SFT (1k)	13	6	4	2	0	0
+ KernelMosaic SFT (full)	64	31	62	51	12	6

Table 7: Composition of KernelMosaic (14,566 tasks). Each task has one development trajectory; train and validation contain disjoint task identifiers.

Input category	Dataset	Training	Validation
Natural language	354	334	20
PyTorch code	9,964	9,444	520
Mixed language and code	1,772	1,683	89
Algebra/matrix	451	430	21
CUDA programming tasks	2,025	1,924	101
Total	14,566	13,815	751

and H20 evaluation GPUs. CUDA Agent is trained from the proprietary Seed1.6 model. Candidate selection maximizes speedup over `torch.compile`; Table 2 reports the eager-reference scores.

Additional performance metrics. Table 6 combines AutoTriton's reported fast$_2$ values with our measured thresholds. ConCuR reports correctness and fast$_1$; CudaForge additionally reports speedup percentiles and geometric means; CUDA Agent reports geometric means and faster-than-compiled coverage.

D Dataset construction and composition

D.1 Task inventory and provenance

Table 7 partitions the 14,566 task–trajectory records from 41 sources by input category and split. Each task has one successful trajectory paired with an implementation that passes its evaluation checks; intermediate failed attempts remain part of the trajectory.

Task records preserve source attribution, interface requirements, and curation decisions. Generation records link candidate revisions, model configurations, selected solutions, and evaluation outcomes. Evaluation records distinguish reference-execution checks, restricted-buffer checks, and candidate tests. Some recovered controls transfer reference outputs to CUDA to exercise the candidate interface.

D.2 Curation parameters

Admission decisions. Individual reviews assess whether a task has a workable CUDA objective and a sufficiently specified computation. Known semantic defects and non-CUDA targets are excluded. The source-group admission policy requires at least 20 reviewed tasks, with more than

80% scoring at least three on a zero-to-four workability scale. Tasks from an upstream validated collection also require verified extraction of their input specifications. Additional references from established libraries reuse source audits meeting the same workability-rate threshold and require an explicit CUDA implementation task. Individual defect findings override source-level admission evidence.

Selection parameters. Global diversity selection uses a cosine-similarity cutoff of 0.925. The priority weight is the product of source, review, category, mathematical-reasoning, and CUDA-relevance factors. Source weights follow fixed provenance profiles; tested library references receive a factor of 1.08. A recorded review score q_i contributes $1 + 0.04 \max(0, q_i - 2)$, with a neutral factor of one for tasks without an individual score. Category factors are 1.10, 1.00, 1.08, 1.12, and 1.02 for natural language, PyTorch code, mixed language/code, algebra/matrix, and CUDA programming, respectively. Mathematical tasks that leave a transformation for the model to discover receive a factor of 1.10, and tasks labeled as primary kernel computations receive 1.02; other tasks receive one for these factors. Ties are resolved by weight, prompt length, and task identifier.

Selection operates on one global embedding pool. A subsequent selection over mathematical tasks uses a cutoff of 0.97, preserving the previously retained set and adding examples without a sufficiently similar retained representative. Category weights express selection preferences rather than fixed quotas.

D.3 Test-suite construction

We use GLM-5.3 (Z.ai, 2026) to generate and repair executable test suites and implementation interfaces from the curated specifications and recovered upstream materials. Generated suites define test inputs, expected outputs, and numerical tolerances. Reference execution and candidate controls provide feedback for repairing test code and interfaces before trajectory generation.

D.4 Trajectory generation

We use Qwen3.8-Flash (Qwen Team, 2026b) with reasoning enabled. The fixed generation policy allocates 100,000 output tokens and 3,600 active seconds for an initial attempt. Eligible continuations use a cumulative ceiling of 300,000 output tokens and up to 7,200 active seconds, retaining the earlier resource consumption. Each response is capped at 32,768 output tokens, and 300 seconds are reserved for finalization.

Evaluation is serialized per GPU to reduce interference, while generation and compilation can proceed concurrently. Compilation results are reused only for matching source and toolchain configurations. Stopping conditions and resource limits are applied at the trajectory level. Failures caused by the execution environment are distinguished from numerical mismatches.

Optional solution explanations. The framework can produce a standalone explanation from the public development history and selected source. This option requires the implementation to pass the public and held-out checks specified for that output format. Hidden outcomes are excluded from explanation generation. The resulting explanation summarizes the selected implementation and its development history. The trajectory-based training representation in this paper is described in Appendix A.1.

D.5 Benchmark exclusion and duplicate checks

To avoid benchmark contamination, we exclude benchmark splits and benchmark-associated repository paths during collection. Comparing the final dataset against all 250 KernelBench tasks yields no duplicates, including after removing comments, docstrings, and formatting differences.

E Dataset analysis

We characterize computational vocabulary, task coverage, development context, supervision, and sensitivity to semantic selection. These analyses distinguish the diversity of specifications from

the distribution of training examples. Related resources likewise examine domain composition, program length, and reasoning length (Heakl et al., 2026; Kong et al., 2025); here, the unit of analysis additionally includes the development trajectory. Input categories follow the reviewed task labels. Trajectory and window analyses pool the training and validation splits.

E.1 Computational vocabulary and co-occurrence

The specifications span neural operations, linear algebra, reductions, tensor organization, and parallel algorithms. Figure 6 provides an overview through 48 computational descriptors, while Figure 7 shows how these groups appear across input forms. Descriptors are matched against the visible task specification using a fixed vocabulary of terms and API names. Each task contributes at most once per descriptor or group, and a task may match several groups. Counts measure explicit mentions in the specification, including comments and constraints. Normalizing within each input form makes the smaller categories visible alongside the larger PyTorch collection.

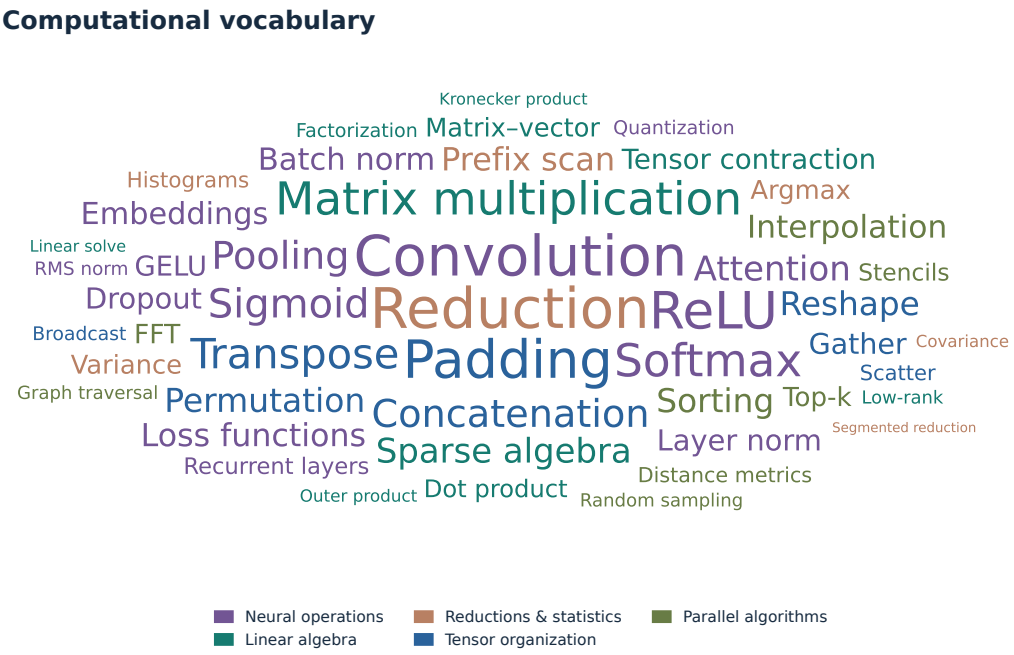


Figure 6: Computational vocabulary across the 14,566 dataset task specifications. Word size increases with the square root of the number of tasks mentioning each descriptor; colors identify five thematic groups. Terms include both natural-language expressions and corresponding API names.

The vocabulary analysis also reveals combinations within individual tasks. Using the same fixed descriptor rules as Figure 6, 6,834 tasks (46.9%) match two or more descriptors, and 5,736 (39.4%) span at least two thematic groups. Multiple descriptors occur in 58.4% of PyTorch tasks and 38.4% of algebra tasks. Figure 8 complements marginal descriptor counts with their co-occurrence structure.

We measure pairwise association by Jaccard overlap: the number of tasks matching both descriptors divided by the number matching either. Convolution and padding co-occur in 1,614 tasks, giving 55.3% overlap. Matrix multiplication and transpose co-occur in 364 tasks (16.9%), while attention and softmax co-occur in 264 (15.1%). These associations characterize combinations of concepts expressed in the task specifications.

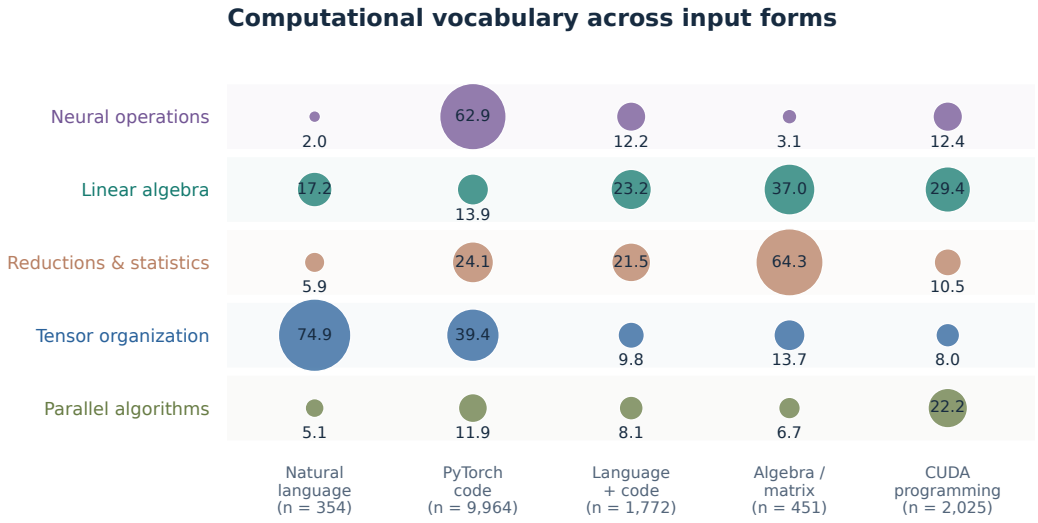


Figure 7: Vocabulary across input forms. Circle area and numerical labels encode the percentage of tasks in each input category that mention at least one descriptor from the corresponding group. Groups may share tasks; each task is counted once per group. Column counts include all dataset tasks, including those without a matched descriptor.

E.2 Input coverage and source profiles

Figure 9 compares input-category proportions and source concentration across the complete dataset and its splits. The training and validation splits have similar category proportions; Table 7 gives the corresponding counts.

Following source-composition analyses in SWE-smith (Yang et al., 2025), we also examine computational vocabulary within each source. Figure 10 separates the five largest sources from the remaining 36, allowing their content profiles to be compared.

The profiles expose distinct concentrations. ReLU and convolution appear in 27.1% and 24.9% of KernelBook tasks, whereas sparse algebra and FFT appear in 22.3% and 10.4% of CASS tasks. Across the remaining sources, transpose and reduction appear in 32.3% and 31.7%. These differences give a concrete view of the complementary content brought together by collection across sources.

E.3 Trajectory length and development depth

Figure 11 reports empirical cumulative distributions over all 14,566 trajectories. The median complete trajectory contains 30,904 tokens and 14 assistant decisions; the 90th percentiles are 79,308 tokens and 31 decisions. In total, 47.4% exceed the 32,768-token training context limit. This distribution motivates segmenting long development trajectories into multiple training windows.

Trajectory length differs by specification category. PyTorch tasks have a median of 38,879 tokens and 18 decisions, compared with 13,203 tokens and 10 decisions for CUDA programming tasks. The difference also affects the number of windows derived from each task. Length measures the recorded development process, including prompts, reasoning, actions, and observations.

Figure 2 in Section 5 relates trajectory length to interaction depth. In the shorter 16,384–32,768-token interval, median decision counts range from nine for natural-language tasks to twelve for PyTorch tasks, complementing the longer-trajectory comparisons in the main text.

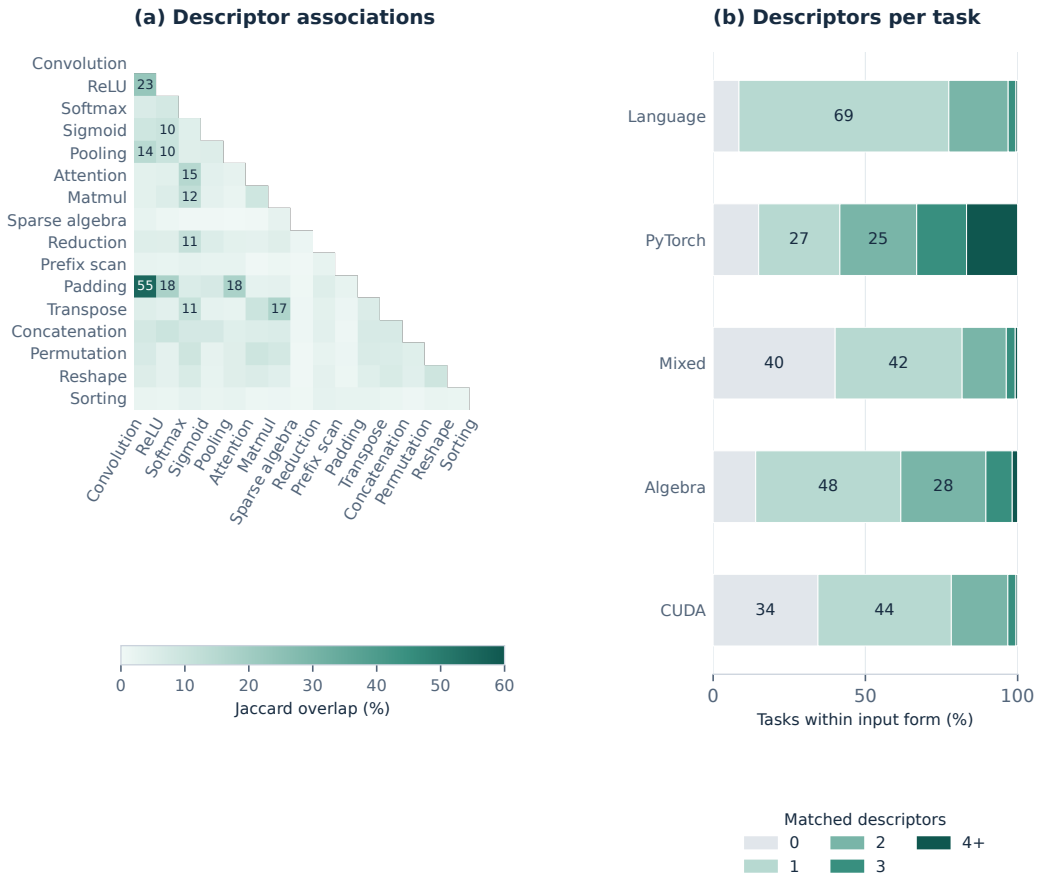


Figure 8: Computational concepts within task specifications. (a) Jaccard overlap among the 16 most frequent descriptors, ordered by thematic group and frequency. Values of at least 10% are labeled; the diagonal is omitted. (b) Number of distinct matched descriptors per task within each input category. The fixed lexical rules are applied to the task text, including code and comments; a zero denotes no match to the descriptor vocabulary. All 14,566 tasks are included.

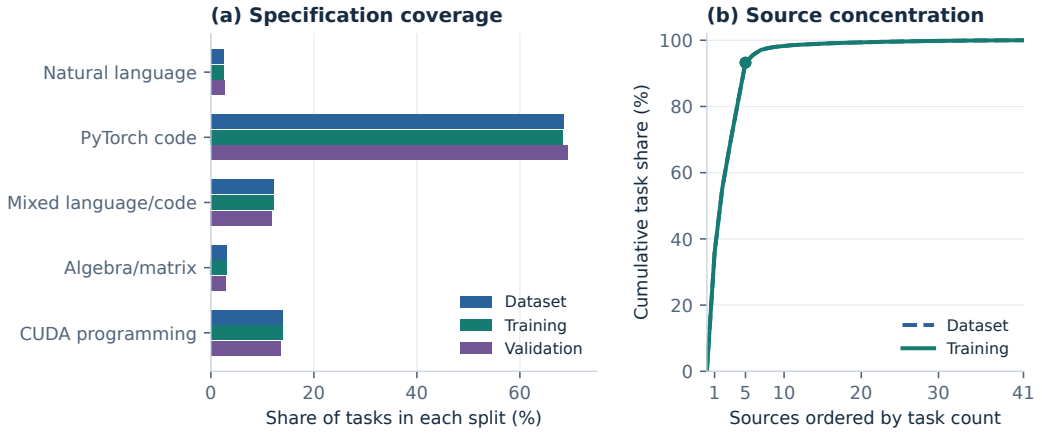


Figure 9: Composition of the 14,566-task dataset. (a) Category proportions in the complete dataset and its training and validation splits. (b) Cumulative task share after independently ranking sources by size in the dataset and training split; markers identify the five largest sources.

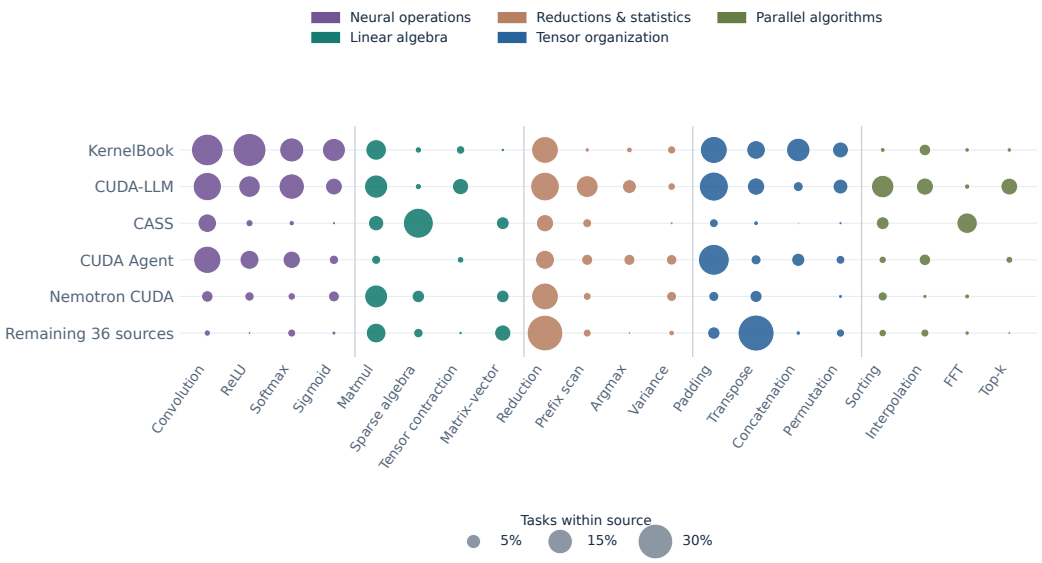


Figure 10: Computational profiles of the collected sources. Circle area represents the percentage of tasks within a source matching a descriptor; colors indicate thematic groups. We display the four most frequent descriptors in each group, selected using the complete dataset. Source rows contain 5,219, 2,839, 1,935, 1,817, 1,772, and 984 tasks, respectively. The final row pools the remaining 36 sources. Source names refer to their contributions retained in KernelMosaic.

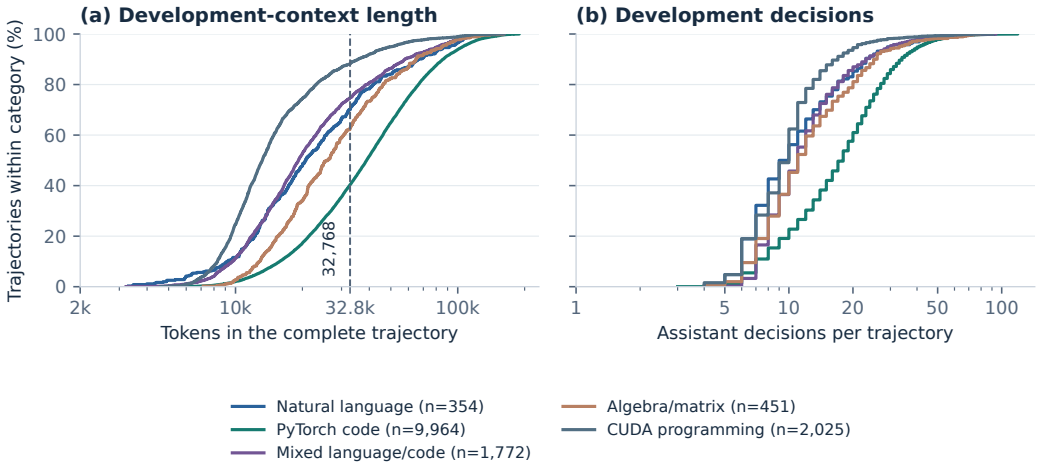


Figure 11: Empirical trajectory distributions by input category, pooling both data splits. (a) Complete tokenized trajectory length before removal of historical reasoning. The dashed line marks the training context limit. (b) Number of assistant decisions, each comprising an assistant message and any associated tool observations. Curves normalize within each category; horizontal axes are logarithmic.

E.4 SUPERVISION AND CONTEXT PRESERVATION

PyTorch tasks contribute 68.4% of trajectories but 80.6% of windows (Figure 12a). They produce 3.17 windows per trajectory on average, compared with 1.37 for CUDA programming tasks. Windowing therefore increases the share of examples from categories with longer trajectories.

Assistant targets comprise 41.5% of all window tokens. The remaining tokens provide task and interaction context, including 26.0% tool observations and 28.4% earlier assistant actions. Removing historical reasoning reduces total tokens by 41.0% relative to restoring that reasoning in the same windows with the same history truncation decisions (Figure 12c). Newly supervised reasoning remains intact. Appendix A.1 specifies the construction and loss mask.

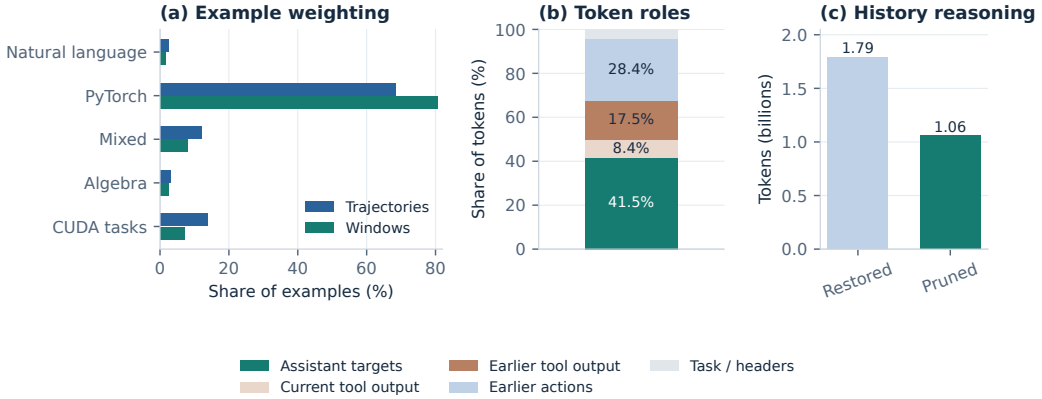


Figure 12: How trajectories become learning examples. (a) Category shares among trajectories and derived windows. (b) Token roles across 1.057 billion tokens in both splits; only assistant targets contribute to the loss. Tool categories include message wrappers; assistant categories include message bodies and end tokens. (c) Token totals with historical reasoning restored or removed, holding windows and history truncation fixed. Token totals are measured per dataset copy.

E.5 SENSITIVITY OF SEMANTIC SELECTION

We examine the category-specific similarity threshold using a recorded algebra-selection sweep over a separate candidate pool before trajectory generation. Holding the previously selected collection fixed, we vary the cosine-similarity cutoff for admitting additional algebra tasks. A higher cutoff retains more candidates by requiring stronger similarity before treating a task as covered. Among 509 candidates, the retained set increases from 339 at 0.925 to 485 at the selected cutoff of 0.97 (Figure 13). The added tasks include both explicit formulations and specifications that leave a useful transformation implicit.

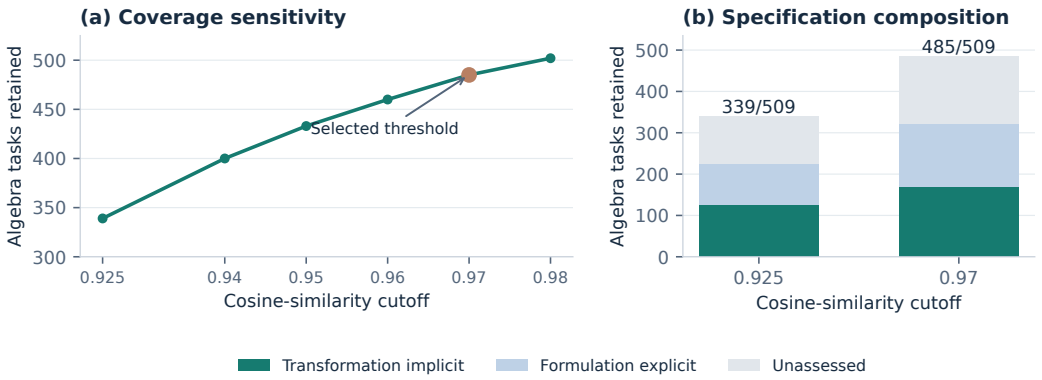


Figure 13: Sensitivity of algebra coverage to the similarity cutoff, with the previously selected collection held fixed. (a) Six measured selection settings. (b) Retained specifications at the default and category-specific cutoffs, grouped by existing review annotations. Unassessed tasks lack this annotation. Counts refer to the pre-generation algebra candidate pool.

F MODEL BEHAVIOR ON KERNELBENCH

We analyze all 500 evaluation trajectories: one from each checkpoint for every L1, L2, and L3 task. The analysis links archived tool actions with the candidate evaluations and resource counters used to compute the benchmark results.

F.1 EVALUATION FAILURES AND CORRECTNESS REPAIR

The original Qwen3.5-9B checkpoint submits candidates for evaluation on 249 of 250 tasks, spanning 99 L1, 100 L2, and all 50 L3 tasks. Across its trajectories, 25,509 file writes complete successfully; calls to tool names outside the provided interfaces account for 97 of 35,323 actions (0.27%).

At the first evaluation, 73 tasks fail because the candidate omits the required `cpp_sources` argument to PyTorch's CUDA-extension loader, and 61 encounter module-state loading errors, including incompatible parameter names and incorrect `load_state_dict` signatures. Another 20 invoke PyTorch tensor computations in place of the required CUDA implementations. Other observed failures include undefined kernel entry points, compilation errors, and incorrect output shapes.

Across complete trajectories, we classify feedback about outputs, execution, and CUDA implementation requirements, extending KernelBench's error analysis (Ouyang et al., 2025). Figure 14 summarizes each task's evaluation responses within the scoring budget. After SFT, correct evaluations account for 64.4% and 53.6% of the mean response distribution on L1 and L2, compared with 0.4% and 0% for the original model. Build or execution errors decrease from 58.2% to 13.7% on L1 and from 52.7% to 23.0% on L2. On L3, their shares remain similar (47.7% and 47.8%), while output mismatches account for 20.2% after SFT. Thus, the change in feedback is strongest on individual operators and fused computations.

The first recorded evaluation passes on 40 L1, 23 L2, and two L3 tasks after SFT. Subsequent revisions solve a further 48, 59, and six tasks, respectively. In every such recovery, the first correct candidate differs from the initially evaluated candidate. Overall, 113 of 178 solved tasks (63.5%) reach correctness through revision; on L2, the proportion is 59 of 82 (72.0%). The original model's four successes also require revision, all on L1.

F.2 TOOL USE AND ACTION SEQUENCES

Each distinct tool call in the message archive contributes one action, including calls that return an error or remain interrupted. Figure 5 first normalizes tool counts within each task and then averages the resulting proportions across tasks. This gives each workload equal weight. The Other category comprises tool names outside the five provided interfaces.

A repeated write is a completed write whose file path and content exactly match an earlier completed write in the same trajectory. We compare source contents by hash and count repetitions after the first occurrence. Table 8 reports the fraction of all completed writes satisfying this criterion. Across the three levels, the original checkpoint repeats 15,322 of 25,509 completed writes (60.1%), compared with 21 of 795 (2.6%) after SFT. The numbers of tasks containing at least one repeated write are 229 and 14, respectively.

Table 8: Trajectory behavior by model and KernelBench level. Calls is the median number of tool calls per task. Repeated writes is the percentage of completed writes reproducing an earlier file path and content. Profiling is the percentage of tasks invoking the profiler.

Model	Level	Calls	Repeated writes	Profiling
Original	L1	178.5	66.5%	6%
Original	L2	99	53.1%	4%
Original	L3	75	46.7%	2%
KernelMosaic SFT	L1	56	2.7%	87%
KernelMosaic SFT	L2	71.5	2.2%	85%
KernelMosaic SFT	L3	74	3.8%	30%



Figure 14: Feedback and recovery under the common evaluation budget. Top: feedback proportions, normalized within each task and then averaged. Output mismatches include shape, value, and finiteness checks; build/execution errors include candidate construction, loading, and runtime failures; implementation-rule failures concern the CUDA execution requirement. Inconclusive checks remain a separate category. Tasks with no eligible evaluation are included explicitly. Bottom: correctness achieved at the first evaluation or after a revision, with all scheduled tasks in the denominator. Evaluations include recorded cache responses and repeated checks.

Table 8 shows fewer tool calls on L1 and L2 alongside reduced repetition and increased profiling.

Action sequences reveal how an agent uses its tools together. Following the trajectory analyses of SWE-agent and SWE-smith (Yang et al., 2024; 2025), we examine transitions between consecutive tool calls and the length of uninterrupted runs of the same tool type. For each current action, we normalize outgoing transitions within each task and average over tasks containing that transition context.

Figure 15 shows a consistent change across difficulty levels. After evaluation, the original model returns to writing a file in 86.2%, 83.5%, and 78.7% of transitions on L1, L2, and L3. After SFT, these shares fall to 7.1%, 6.2%, and 4.2%; editing becomes the most common next action, followed by code inspection on L1 and L2. The median longest run of a single tool type decreases from 92 to 5 calls on L1, from 16 to 5 on L2, and from 8.5 to 6 on L3. Across all levels, 167 original-model trajectories contain a run of at least ten calls to the same tool, compared with 16 SFT trajectories. These sequence-level measurements complement the aggregate tool proportions: SFT organizes development around shorter sequences of modifications, inspection, and evaluation.

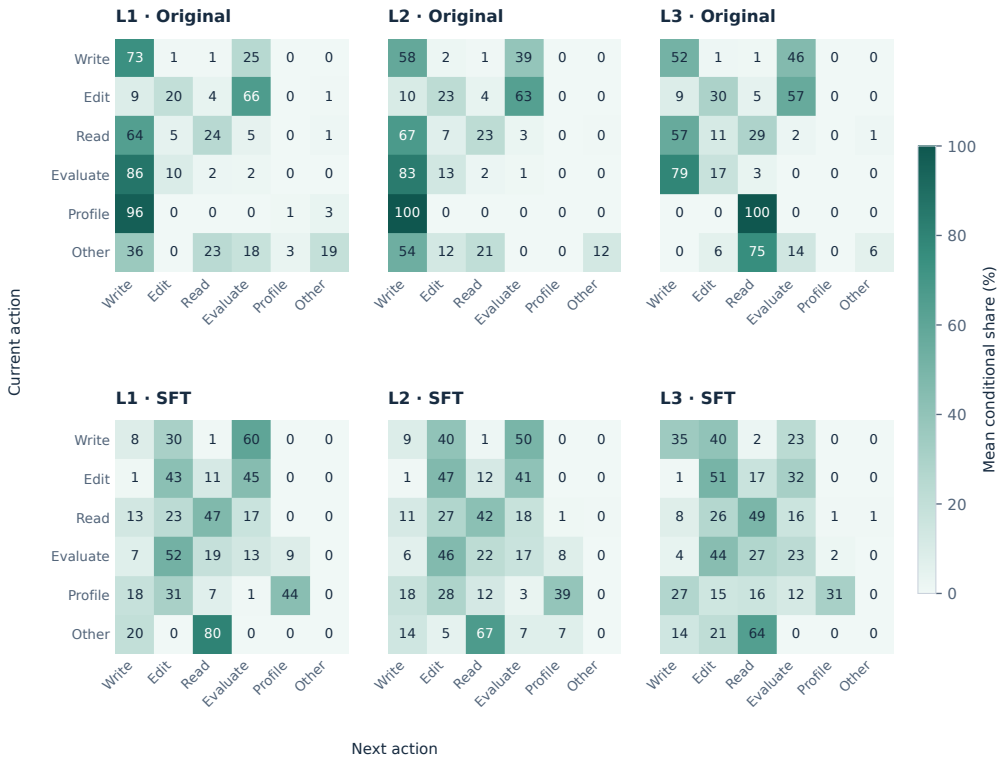


Figure 15: Action transitions in original and SFT trajectories. Rows denote the current tool and columns the next tool; cells show percentages averaged over tasks after row normalization within each task. All archived tool calls are included, ordered by recorded start time. Each row averages only tasks with at least one outgoing transition from that tool. A dash denotes an absent context. Repetition here refers to tool type, whereas Table 8 measures exact repeated write contents.

F.3 Candidate quality and subsequent optimization

We define the first candidate as the implementation submitted at the first recorded evaluation; the agent may reason and edit before that submission. Table 9 reports its correctness alongside the performance of the earliest passing implementation within the evaluation budget. These first correct implementations already outperform eager execution on 101 SFT tasks.

For each solved task, Figure 16 compares the first timed correct candidate with the best candidate found within budget, ordered by recorded active time. Further revisions bring an additional 15 L1

Table 9: First-candidate correctness and first-correct-candidate performance on KernelBench. Correctness and $fast_1$ use all tasks at each level (100/100/50). Solved counts tasks with a passing candidate within budget. Median speedup is computed over each row's solved tasks, using their first passing implementation; a dash denotes no solved tasks.

		First candidate		First correct candidate	
Model	Level	Correct (%)	Solved	Median speedup	$fast_1$ (%)
Original	L1	0	4	0.19×	0
Original	L2	0	0	—	0
Original	L3	0	0	—	0
KernelMosaic SFT	L1	40	88	1.04×	49
KernelMosaic SFT	L2	23	82	1.17×	46
KernelMosaic SFT	L3	4	8	1.17×	12

and 16 L2 tasks above the eager baseline. All six faster-than-eager L3 solutions already cross the baseline at their first timed correct candidate. The original model's four correct candidates remain slower than eager execution.

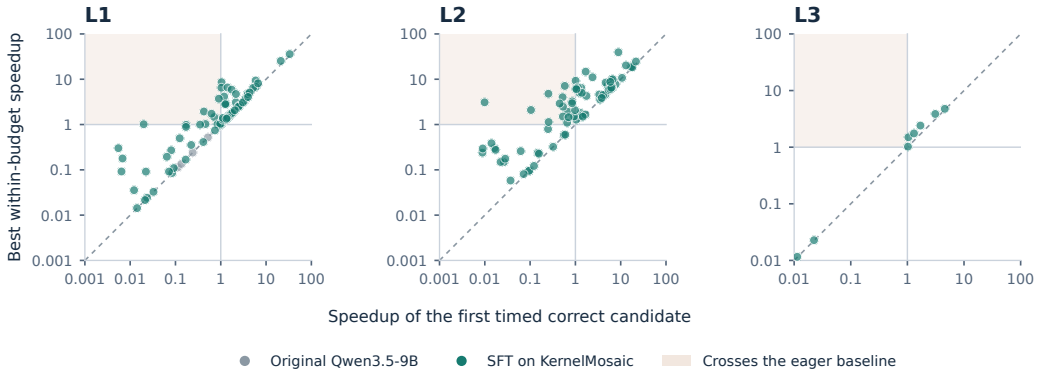


Figure 16: Optimization after the first correct implementation. Each point compares a task's first timed correct candidate with its best measured candidate within the trajectory budget. Both axes show speedup over PyTorch eager on logarithmic scales; diagonal guides mark equal performance. The shaded region contains tasks that cross from slower-than-eager to faster-than-eager implementations. All 178 solved SFT tasks and four solved original-model tasks are included.

F.4 Progress under token and time budgets

We apply output-token and active-time cutoffs to recorded candidate evaluations using the eligibility rules in Appendix B. Correctness and $fast_1$ are computed over all scheduled tasks at each level. Figure 5 varies the token cutoff with the time allowance fixed at 3,600 seconds; Figure 17 varies both resources for the SFT trajectories.

At 65,536 output tokens, SFT solves 66%, 59%, and 8% of L1, L2, and L3 tasks, with faster-than-eager coverage of 49%, 42%, and 8%. At 131,072 tokens, correctness reaches 79%, 73%, and 12%, and $fast_1$ reaches 60%, 55%, and 8%. The original checkpoint solves one L1 task by 65,536 tokens and three by 131,072; its four solved tasks at the full budget are all in L1.

At 450 active seconds, L1 correctness reaches 53% with 131,072 output tokens and remains unchanged at 262,144 tokens. Increasing the active-time allowance to 3,600 seconds at the latter token allowance raises correctness to 88%. Faster-than-eager coverage continues to grow after many tasks have reached correctness: on L1, raising the active-time allowance from 900 to 3,600 seconds at the full token allowance increases correctness from 75% to 88% and $fast_1$ from 47% to 64%.

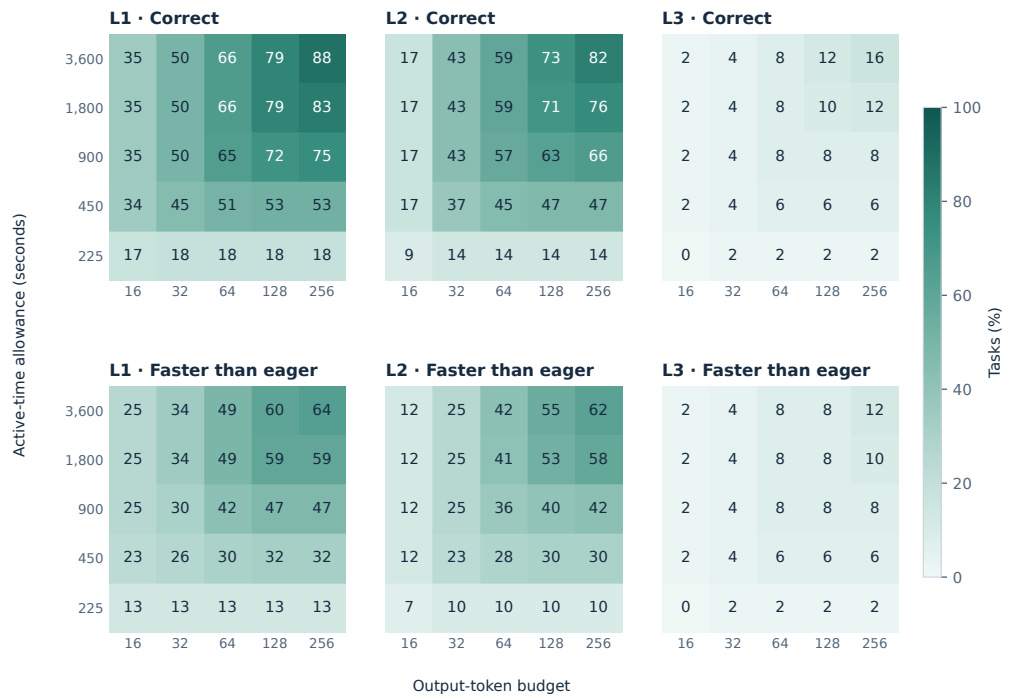


Figure 17: SFT progress under joint resource cutoffs. Cells show the percentage of all tasks with a correct candidate (top) or a trusted faster-than-eager candidate (bottom) recorded before both cutoffs. Horizontal ticks denote 1,024 output tokens; active time excludes inference and infrastructure waits. All panels share the color scale.

F.5 Development activity after one epoch

Across their 250 benchmark trajectories, Direct SFT and Trajectory SFT produce 1,133 and 29,253 archived tool calls, respectively (Figure 18). Median counts are 4, 4, and 3 for Direct SFT on L1–L3, versus 95.5, 114.5, and 94 for Trajectory SFT. Evaluation occurs on 55%, 52%, and 36% of Direct SFT tasks, versus 98%, 92%, and 84% after trajectory supervision; editing occurs on 18%, 11%, and 8%, versus 98%, 93%, and 86%. Trajectory supervision supports sustained development across all three levels.

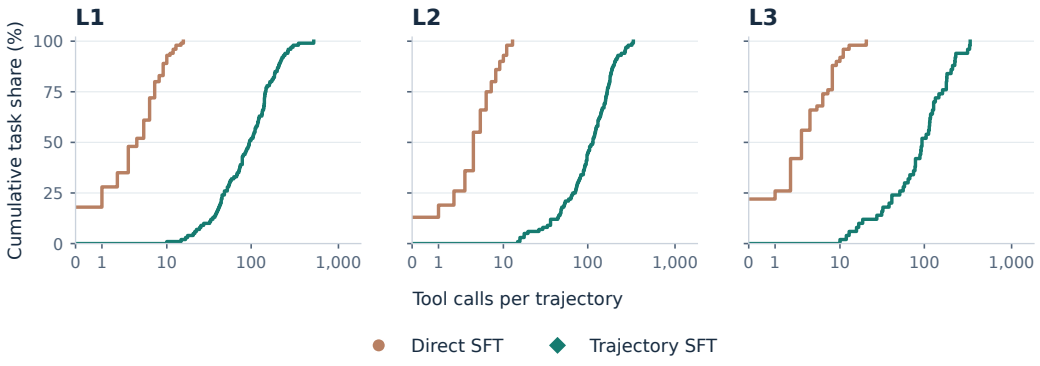


Figure 18: Development activity after one epoch of SFT on 1,000 examples. Curves show cumulative task shares by the number of tool calls, including errored or interrupted invocations. Horizontal ticks show call counts on a $\log(1 + \text{calls})$ scale. Each level includes all scheduled tasks.

G ILLUSTRATIVE DATASET AND DEVELOPMENT EXAMPLES

These five examples illustrate the dataset's input forms. Descriptions are abridged and lightly paraphrased, preserving the computational target and material constraints while omitting boilerplate and supporting files. Table 10 records their origins.

Table 10: Illustrative prompts spanning the five curation categories. Input form is distinct from the requested implementation and from the evaluation interface.

Input category	Example	Source / construction origin
PyTorch code	Affine layer normalization	KernelBook (Paliskara & Saroufim, 2025)
Mixed language and code	Floating-point histogram	Nemotron SFT CUDA v1
Natural language	Integer matrix transpose	fhrozen Stack CUDA prompts
Other CUDA code	Missing transform kernel	llama.cpp CUDA backend
Algebra / matrix	Batched low-rank operator	Synthetic algebra construction

G.1 A PYTORCH REFERENCE: AFFINE LAYER NORMALIZATION

The task requests an optimized CUDA replacement that preserves a PyTorch module's behavior and interface, returned as one Python module with CUDA bindings. Its forward computation is:

```
mean = x.mean(-1, keepdim=True)
s = (x - mean).pow(2).mean(-1, keepdim=True)
x = (x - mean) * torch.rsqrt(s + self.eps)
return self.scale * x + self.shift
```

The module initializes a learned scale to ones, a learned shift to zeros, and uses $\epsilon = 10^{-5}$. The supplied input example has shape $(4, 4, 4, 4)$ and four affine features. The computational specification comes primarily from the program; the wrapper states the target language and deliverable. Other shapes used in the development example below belong to its separately constructed adapter.

G.2 LANGUAGE AND A HEADER: A FLOATING-POINT HISTOGRAM

This task from Nemotron-SFT-CUDA-v1 (NVIDIA, 2026) asks for configurable bins over a supplied minimum/maximum range, block-local accumulation in shared memory, and atomic accumulation into global memory, with support for arbitrary element and bin counts. The supplied header declares:

```
void compute_histogram(const float* input, int* histogram,
                       int num_elements, int num_bins,
                       float min_val, float max_val);
```

The requested deliverable is a CUDA source file using the supplied headers without modification. Here prose supplies requirements that the signature alone cannot express, while the header fixes the interface.

G.3 A LANGUAGE SPECIFICATION: INTEGER MATRIX TRANSPOSE

This prompt from the CUDA subset of Stack-Prompts (Fhrozen, 2026) specifies a kernel named `naiveGPUTranspose` with integer input/output pointers and `rows`, `cols` arguments. Each thread obtains a two-dimensional position (i, j), checks $i <$ `rows` and $j <$ `cols`, and writes

$$\texttt{d_b}[j \cdot \texttt{rows} + i] = \texttt{d_a}[i \cdot \texttt{cols} + j].$$

The original specification explicitly requests one element per thread and standard block/thread indexing. Thus the collection includes implementation-constrained instructions as well as open-ended optimization tasks.

G.4 A SOURCE-COMPLETION TASK: A WALSH–HADAMARD TRANSFORM

The source from llama.cpp's CUDA backend (llama.cpp Contributors, 2026) leaves the definition of `fwht_cuda<N>` missing while preserving its signature, launch constraints, and caller.

The requested output is the complete missing definition. The context supplies contiguous tensors of equal shape, row counts, four rows per block, a scale factor $1/\sqrt{N}$, and specializations for $N \in \{64, 128, 256, 512\}$. The kernel receives `src`, `dst`, `n_rows`, and `scale`. Existing helper conventions constrain the reconstruction. This example represents integration with an existing CUDA codebase, where correctness depends on both the transformation and its surrounding call contract.

G.5 A MATHEMATICAL SPECIFICATION: A BATCHED LOW-RANK OPERATOR

The task supplies contiguous, row-major float32 tensors $U, V \in \mathbb{R}^{G\times Q\times R}$ and $X \in \mathbb{R}^{G\times Q\times C}$. For each block b, it defines

$$L_b = U_b V_b^\top, \qquad Y_b = L_b X_b,$$

and requests a float32 output $Y \in \mathbb{R}^{G\times Q\times C}$. All dimensions are positive, $C \geq 2$, and inputs must remain unchanged. Accumulation uses float32 or higher precision, with elementwise absolute and relative tolerances of 10^{-4} against the float64 specification. The requested module exposes `solve(U, V, X, *, dimensions)` and uses custom CUDA kernels through PyTorch bindings. The prompt defines the operator without prescribing the multiplication order. This leaves implementation choices available while making types, shapes, and numerical obligations explicit.

G.6 AN OBSERVED IMPLEMENTATION REVISION

A separate five-task inspection run used Qwen3.8-Flash to develop implementations for these examples. The run used an example-specific adapter with 32 public and 25 hidden input cases across the five tasks. All five selected implementations passed that adapter's public and hidden checks.

Initial implementation. Round 0 assigns one block to each row, with a fixed 256-thread block. It computes the mean, then the centered sum of squares, then normalizes and applies the affine parameters. Two separate block reductions compute the mean and variance, with FP32 accumulation. This version passed all three public cases.

Feedback and revision. The next-round record uses public timing and source inspection to hypothesize that small rows incur avoidable reduction and synchronization overhead. Round 1 jointly accumulates the sum and sum of squares, reduces both quantities together, and computes

$$\mu = \frac{1}{N}\sum_i x_i, \qquad \widehat{\sigma}^2 = \max\left(0, \frac{1}{N}\sum_i x_i^2 - \mu^2\right).$$

It chooses a warp-multiple block size, capped at 256 threads, from the row length. The binding also adds explicit contiguity checks before materialization. The joint reduction uses two synchronization barriers. Table 11 gives the recorded outcomes, including hidden evaluation of the selected revision.

Table 11: LayerNorm implementation revision. Latencies are task-level aggregates from the adapter, in microseconds. Public rounds use the same public cases; the last row uses separate hidden cases.

Candidate / evaluation	Correctness	Candidate (μs)	Eager (μs)	Speedup
Round 0 / public	3/3 cases	16.77	72.28	4.31×
Round 1 / public	3/3 cases	10.76	48.94	4.55×
Selected round 1 / hidden	5/5 cases	11.83	48.95	4.14×