NKI-Agent: Domain-Specific Fine-Tuning and Agentic Tool Use for Neuron Kernel Generation

Junjie Tang [1] **Jun Huan** [1] **Hao Zhou** [1] **Yuhao Zhang** [1] **Lin Wang** [1]

Abstract

Recent agentic approaches to LLM-based kernel generation have achieved strong results on CUDA, yet emerging AI accelerators such as AWS Trainium and Inferentia remain unaddressed. Writing kernels for these chips via the Neuron Kernel Interface (NKI) is particularly challenging due to a multi-engine architecture, tile-based programming with a fixed 128-element partition dimension, and explicit memory management. Moreover, no training data, benchmarks, or tool-augmented agents exist for this domain. We introduce NKI-AGENT, the first system combining domain-specific supervised fine-tuning (SFT) with a compile-verify-fix agent loop for NKI kernel generation. We adapt the existing CUDA-Agent framework to Neuron hardware, curate 6,000 NKI kernel generation tasks for training, and construct NKIGEN-BENCH, a 250-task benchmark across three difficulty levels. Evaluated on real Trn1 hardware, NKI-AGENT with Claude Opus 4.8 and a rank-aware system prompt achieves a **77.3% pass rate** on the 150-task NKIGEN-BENCH (63.3% on a 60-task subset), while our SFT-trained Qwen3-Coder-30B-A3B achieves 25.0% pass rate (60-task) at **1/100th the cost**, outperforming Claude Sonnet 4 (15.0%) with identical tools. Tool use proves essential: even Opus 4.8 scores 0% in single-shot mode on the 60-task subset (6% on 150-task) versus 77.3% with agent tools. We also report that Group Relative Policy Optimization (GRPO) with binary compilation reward fails to improve over SFT, providing guidance on reward design for RL-based kernel generation.

[1]Amazon Web Services. Correspondence to: Junjie Tang <junjtang@amazon.com>.

Proceedings of the 43^{rd} International Conference on Machine Learning, Seoul, South Korea. PMLR 306, 2026.

1. Introduction

The rapid scaling of deep learning models has intensified demand for hardware-specific kernel optimization. While NVIDIA GPUs have benefited from decades of CUDA ecosystem development and compiler optimizations, emerging AI accelerators such as AWS Trainium and Inferentia require new approaches to kernel generation. The Neuron Kernel Interface (NKI) (AWS Neuron Team, 2024a) provides a Python-based programming model for these chips, but writing high-performance NKI kernels remains challenging due to the unique multi-engine architecture and tile-based programming model.

Recent work on agentic code generation has demonstrated that language model agents equipped with execution feedback can generate competitive GPU kernels. The CUDA-Agent (Wu et al., 2026) system uses Proximal Policy Optimization (PPO)-trained agents on a synthesized dataset of 6,000 CUDA tasks, achieving 92–100% faster rates over `torch.compile` on KernelBench (Ouyang et al., 2025). However, this approach is specific to CUDA and does not address the distinct challenges of Neuron hardware.

NKI programming presents unique challenges: (1) **Multi-engine architecture** with four specialized compute engines (Tensor, Vector, Scalar, GpSimd); (2) **Tile-based programming** on 2D tiles with fixed 128-element partition dimension; (3) **Explicit memory hierarchy** requiring manual data movement between HBM (high-bandwidth memory), SBUF (on-chip scratchpad), and PSUM (partial-sum accumulator); (4) **Evolving SDK** with API changes between versions.

In this paper, we present NKI-AGENT, the first system to apply domain-specific fine-tuning and agentic tool use to NKI kernel generation. Our contributions:

1. **NKI-AGENT**: A multi-turn agent with compile and verify tools and a rank-aware system prompt for NKI kernel generation, trained with supervised fine-tuning (SFT) and Group Relative Policy Optimization (GRPO) on Qwen3-Coder-30B-A3B.
2. **NKI-Agent-Ops-6K & NKIGEN-BENCH**: 6,000 curated NKI tasks and a 250-task benchmark across three difficulty levels.

3. **SFT data quality analysis**: Four dataset iterations showing data curation matters more than hyperparameters.

4. **Honest negative result**: GRPO with binary reward fails to improve over SFT.

The NKI-AGENT framework achieves 77.3% on the 150-task NKIGEN-BENCH with a frontier model (Opus 4.8 with a rank-aware prompt), and our domain-specialized SFT model captures a large fraction of this performance at 1/100th the cost (25.0% vs 63.3% on the 60-task subset), outperforming Sonnet 4 (15.0%) with identical tools.

2. Related work

LLM-based kernel generation. KernelBench (Ouyang et al., 2025) benchmarks LLM-generated GPU kernels. CUDA-Agent (Wu et al., 2026) trains PPO-based agents on 6,000 CUDA tasks with compile-verify-profile tools. Other approaches include AlphaCode-style search (Li et al., 2022), self-repair (Olausson et al., 2024), and TritonBench (Various, 2025). Related code-translation work generates accelerator code by translating across languages: TransCoder (Roziere et al., 2020), BabelTower (Wen et al., 2022), and CodeRosetta (TehraniJamsaz et al., 2024). The agentic-coding evaluation lineage—SWE-bench (Jimenez et al., 2024) and SWE-agent (Yang et al., 2024)—motivates our tool-augmented, execution-grounded setup.

RL for code generation. CodeRL (Le et al., 2022) and PPOCoder (Shojaee et al., 2023) use execution rewards; DeepSeek-R1 (DeepSeek-AI, 2025) demonstrates GRPO (Shao et al., 2024) without a critic. CUDA-Agent shows RL with graded reward outperforms SFT; we show the converse for binary reward.

Neuron hardware and NKI. AWS Trainium/Inferentia chips (Amazon Web Services, 2024) feature four specialized compute engines. NKI (AWS Neuron Team, 2024a) provides tile-based programming; prior work is limited to reference kernels (AWS Neuron Team, 2024b). AccelOpt (Zhang et al., 2026) (concurrent) addresses NKI kernel *optimization*, complementary to our correctness-first generation.

3. Method

Figure 1 illustrates the NKI-AGENT system. It consists of four components: (1) a data synthesis pipeline producing 6,000 curated NKI tasks, (2) a multi-turn agent with compile and verify tools, (3) a binary reward function, and (4) a two-stage SFT→GRPO training pipeline.

3.1. NKI programming model

Neuron chips feature four specialized engines: **Tensor** (matrix ops), **Vector** (elementwise/reduction), **Scalar** (control flow), and **GpSimd** (general-purpose). Programs operate on 2D tiles with a fixed 128-element partition dimension. Data must be explicitly moved between HBM (device memory), SBUF (24 MB scratchpad), and PSUM (partial sum accumulator) in a load→compute→store pattern. This differs fundamentally from CUDA's thread-block model, requiring domain-specific knowledge for correct kernels.

3.2. Data synthesis pipeline

We synthesize NKI-Agent-Ops-6K in three stages: (1) **Seed crawling** from PyTorch `torch.nn` (∼180 ops), `nki-samples` (∼30 kernels), and KernelBench (∼250 tasks), yielding ∼400 seeds. (2) **Combinatorial expansion** via shape (5×), dtype (3×), and fusion synthesis (∼200 patterns), producing ∼15K candidates. (3) **Filtering** through compilation, execution, stability (CV<10%), and baseline checks, balanced to 6,000 tasks. The 250 NKIGEN-BENCH tasks are held out. For SFT, we curate 647 high-quality episodes through four iterations (Section 5.1).

3.3. Agent architecture

NKI-AGENT operates through multi-turn interaction with two tools on real Trn1 hardware: (1) **compile_kernel** invokes `neuronx-cc` and returns compilation status with error messages (timeout: 120s); (2) **verify_kernel** runs the compiled kernel with random inputs, comparing against the PyTorch reference via `torch.allclose` (atol=rtol=10^{-3}, 3 trials). The agent iterates up to $T=10$ turns: generate→compile→verify→fix. We set $T=10$ empirically: in our evaluation runs the vast majority of eventual successes converge within the first few turns, and gains beyond ten turns are negligible while the per-turn Trn1 compile/verify cost grows linearly, so $T=10$ balances solution coverage against evaluation cost.

Rank-aware system prompt. For frontier models we use a *rank-aware* system prompt. NKIBench inputs span 2-D, 3-D, and 4-D tensors (e.g. batched attention and N,C,H,W convolutions), but a naive prompt assumes a 2-D (M, N) layout. The rank-aware prompt instructs the model to read each task's input-shape block and provides templates for 2-D, 3-D, and 4-D kernels, plus a reminder that every output element must be written by an `nl.store`. This is a prompt-only change—no retraining or extra tool calls—and is the primary driver of the frontier model's Level 2/Level 3 gains (Section 4.4).

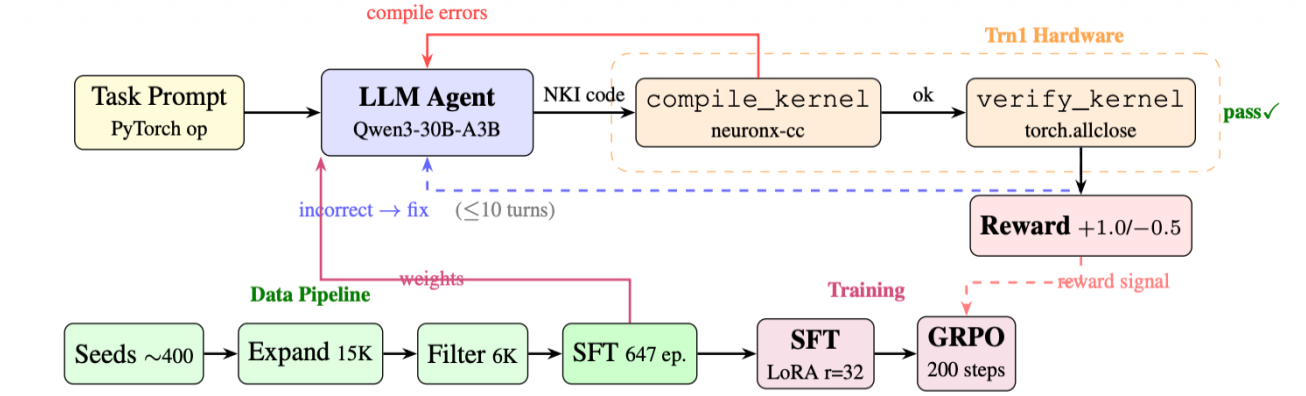

Figure 1. **NKI-AGENT system overview.** *Top:* The agent generates NKI code, then iteratively compiles and verifies on Trn1 hardware ($\leq$10 turns). Compile errors and verification failures feed back for correction. *Bottom:* Data pipeline produces 6,000 tasks; SFT on curated episodes followed by GRPO with binary reward.

3.4. Training pipeline

Stage 1: SFT. We fine-tune Qwen3-Coder-30B-A3B (Qwen Team, 2025) on 647 curated episodes using LoRA (Hu et al., 2022) (rank 32, alpha 64, 26.7M params / 0.09%). Training: 2 epochs, lr=2×10^{-4}, batch 4, BF16 on 8×A100 40GB.

Stage 2: GRPO. From the SFT checkpoint, we apply GRPO (Shao et al., 2024). For each prompt, G=4 kernels are generated, each scored with binary reward ($+1.0$ correct, -0.5 fail), and group-relative advantages computed. Training: 200 steps, lr=5×10^{-7}, β=0.1, ~18h on 8×A100 40GB.

Base model. We selected Qwen3-Coder-30B-A3B (Qwen Team, 2025) after evaluating models from 1.1B to 30B parameters. Smaller models (TinyLlama 1.1B, Mistral 7B) failed to generate syntactically valid NKI code even after SFT. Qwen3-Coder's MoE architecture (30B total, 3B active) balances capability with inference speed (7.1 tok/s) for multi-turn agent interaction. We serve via vLLM with tensor parallelism and native function-calling.

Table 1. **Main results on NKIGEN-BENCH (150-task).** Pass rate on balanced subset (50 per level). All results from real Trn1 hardware. Note: at N=150, differences of 1–2 tasks (~0.7pp) are within noise; per-level and 60-task results (Tables 2, 3) show clearer separation.

Model	Single-Shot	NKIAgent (10 turns)
Base (Qwen3-30B-A3B)	14.0% (21/150)	20.0% (30/150)
SFT v4	19.3% (29/150)	**20.7% (31/150)**
GRPO	15.3% (23/150)	17.3% (26/150)

4. Experiments

4.1. NKIGEN-BENCH

NKIGEN-BENCH[1] contains 250 tasks across three difficulty levels: **Level 1** (100 tasks: single operations—matmul, activations, reductions, convolutions), **Level 2** (100 tasks: fused operations—conv+relu+bias, attention blocks, MLP components), and **Level 3** (50 tasks: full model components—Transformer layers, ResNet bottlenecks, Mamba blocks). Each task provides a PyTorch `Model` class and NKI-specific metadata. Of 250 tasks, ~150 are adapted from KernelBench, 60 require decomposition, and 40 are NKI-native. We evaluate on a balanced 150-task subset (50 per level) and a 60-task subset for ablations.

[1]Not to be confused with the concurrently developed NKIBENCH of Zhang et al. (2026), a kernel *optimization* suite for Trainium; NKIGEN-BENCH targets kernel *generation* and correctness.

Table 2. **Pass rate by difficulty level** (150-task, 50 per level). NKIAgent tool use uniquely enables Level 2 and Level 3 tasks where single-shot achieves near-zero success. Opus 4.8 uses the rank-aware system prompt.

	Single-Shot			NKIAgent		
Model	L1	L2	L3	L1	L2	L3
Base	34%	8%	0%	26%	18%	16%
SFT v4	**50%**	8%	0%	30%	16%	16%
GRPO	44%	2%	0%	22%	18%	12%
Opus 4.8[†]	10%	8%	0%	**84%**	**74%**	**74%**

[†]Rank-aware system prompt; SS (10%/8%/0%) shown for reference.

4.2. Evaluation setup

Models: (1) Base Qwen3-Coder-30B-A3B; (2) SFT v4 (647 episodes); (3) GRPO (SFT v4 + 200 RL steps); (4) Claude Sonnet 4; (5) Claude Opus 4.8 (frontier upper bound). All Claude models use identical NKI-AGENT tools. Frontier results use a rank-aware system prompt (Section 3.3). **Modes:** Single-shot (SS, no tools) and NKIAgent ($\leq$10 turns with tools). **Metric:** pass rate = compiles on `neuronx-cc` + numerically correct (atol=rtol=10^{-3}) vs PyTorch reference. All on real Trn1 hardware.

4.3. Main results

Tables 1 and 2 present the main results. On the aggregate 150-task metric, SFT v4 NKIAgent (20.7%) and Base NKIAgent (20.0%) are within statistical noise (1 task apart). The clearer signal emerges from per-level breakdown and 60-task ablations (Table 3). Key findings:

- **SFT v4 + NKIAgent** achieves 20.7% on 150-task and 25.0% on 60-task. The NKI-AGENT framework scales to 77.3% with Opus 4.8 (Table 3), showing the framework design is sound and model capability is the primary bottleneck.
- **Tool use uniquely enables L2/L3.** No model solves any L3 task in single-shot (0/50); NKIAgent achieves 16% via iterative compile-fix.
- **SFT improves L1**: 34%→50%, reflecting learned NKI patterns (tile dims, `nl.load`/`nl.store`).
- **L1 NKIAgent < L1 SS**: a counterintuitive pattern where tool use *hurts* easy tasks. The agent sometimes "fixes" already-correct code after seeing unrelated compiler warnings, or exhausts turns exploring alternatives. Tool use is net-positive only for tasks that *require* iterative debugging (L2/L3).
- **GRPO degrades performance** vs SFT v4 across all settings (Section 5.2).

Table 3. **Ablation study** comparing evaluation modes and model configurations. The 60-task subset enables additional comparisons. "—" indicates configurations not evaluated on the larger set due to compute constraints (each 150-task NKIAgent run requires ∼20 hours on Trn1). **Bold**: best trained model and best overall.

Configuration	**60-task**	**150-task**
Base SS	10.0% (6/60)	14.0% (21/150)
SFT v4 SS	15.0% (9/60)	19.3% (29/150)
SFT v4 MT-5 (no tools)	15.0% (9/60)	—
Base NKIAgent	16.7% (10/60)	20.0% (30/150)
SFT v4 NKIAgent	**25.0% (15/60)**	**20.7% (31/150)**
GRPO NKIAgent	16.7% (10/60)	17.3% (26/150)
Sonnet NKIAgent	15.0% (9/60)	—
Opus 4.8 SS	0.0% (0/60)	6.0% (9/150)
Opus 4.8 NKIAgent	**63.3% (38/60)**	**77.3% (116/150)**

4.4. Single-shot vs. NKIAgent

Table 3 reveals several findings. Multi-turn without tools does *not* help (MT-5 = SS at 15.0%), echoing the Reflexion (Shinn et al., 2023) finding that verbal self-feedback without grounded execution signal yields limited gains. SFT and NKIAgent are super-additive (+5.0pp + +6.7pp separately, +15.0pp combined). Most strikingly, **Opus 4.8 achieves 0% in single-shot but 63.3% with NKIAgent tools** on the 60-task subset (6% vs 77.3% on 150-task)—the largest tool-use gap in our study, demonstrating that even frontier models *cannot* generate correct NKI kernels without iterative compiler feedback. Opus 4.8's L3 pass rate (74% on 150-task) is particularly notable: where all other models achieve $\leq$16%, Opus 4.8 succeeds on nearly three-quarters of full model components.

Cost-capability tradeoff. Our SFT model (3B active params, 7.1 tok/s) captures 40% of Opus 4.8's performance (25.0% vs 63.3% on the 60-task subset) at ∼1/100th the per-token cost. The SFT model outperforms Sonnet (25.0% vs 15.0%) despite ∼300× less NKI training data than CUDA, demonstrating that domain SFT on scarce data can close a large fraction of the gap to frontier models.

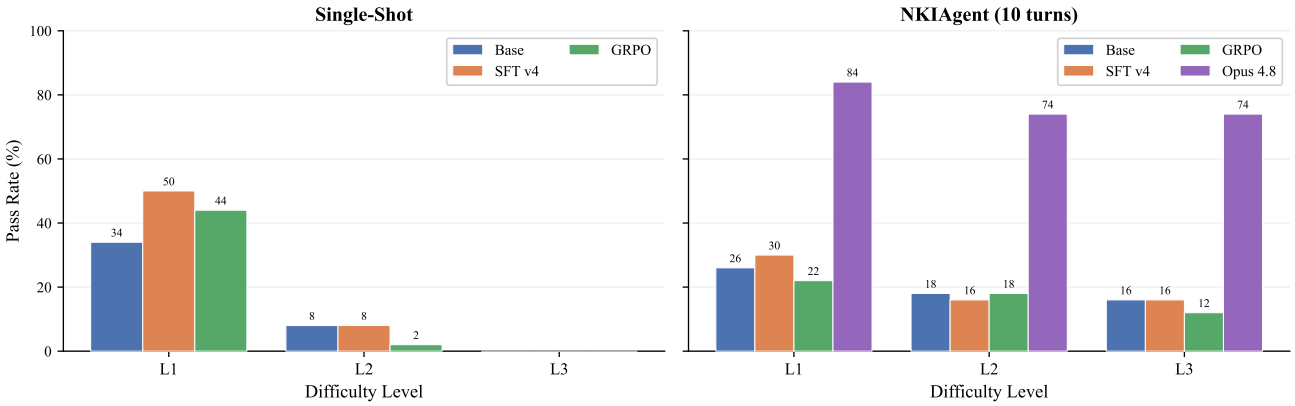

Figure 2. **Pass rate by difficulty level** (150-task). Left: single-shot shows steep degradation from L1 to L3, with all trained models at 0% on L3. Right: NKIAgent enables meaningful L2/L3 performance. Opus 4.8 with the rank-aware prompt (purple) dominates all levels, particularly L3 (74% vs $\leq$16% for trained models).

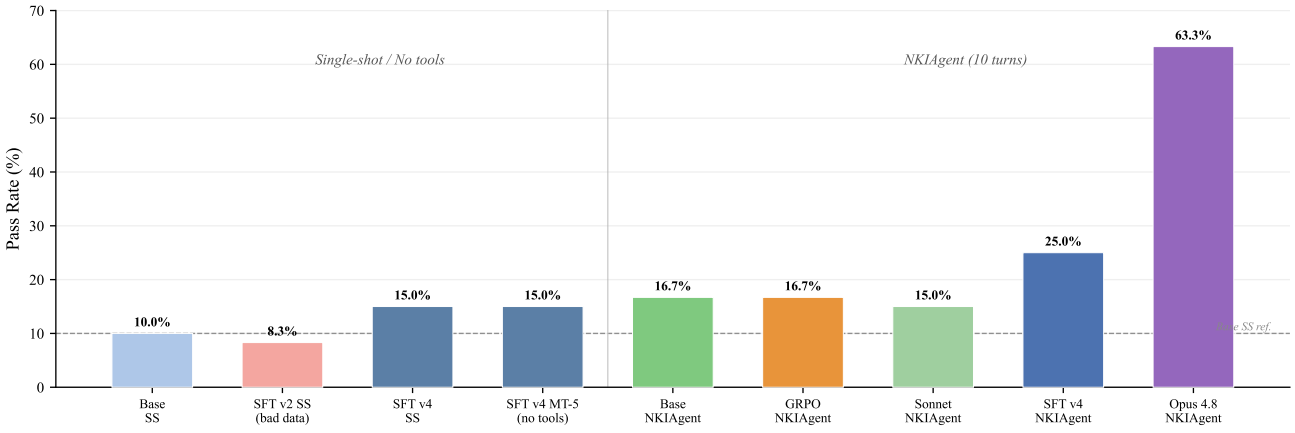

Figure 3. **Ablation study (60-task subset).** Opus 4.8 NKIAgent (63.3%) establishes the framework's upper bound. SFT v4 NKIAgent (25.0%) captures 40% of frontier performance. SFT v2 with bad data (8.3%) underperforms the base model (10.0%).

5. Analysis

5.1. SFT data quality progression

Table 4 traces our data curation journey. SFT v2 with bad data hurts performance (8.3% vs base 10.0%). Each fix targeted one issue: v1→v2 fixed message format; v2→v3 standardized function names (64% were wrong); v3→v4 converted 3D tensors to 2D. Lesson: **each data quality fix produced larger gains than any hyperparameter change**.

5.2. GRPO negative result

Table 5 shows GRPO consistently underperforms SFT v4 (average reward flat at ∼10–15% compile rate across 200 steps). We hypothesize: (1) binary reward is too sparse—no signal between "almost compiles" and "completely wrong"; (2) subtle code changes flip reward discontinuously; (3) when ∼75% of generations fail, group-relative ranking selects among incorrect outputs. CUDA-Agent's graded reward (speedup measurements) provides richer signal. Lesson: **binary reward is insufficient for RL on kernel generation**.

5.3. Error analysis and qualitative findings

Category-level patterns. SFT excels at elementwise (87.5%) and activation (90.0%) operations. Convolutions, attention, and all Level 3 components achieve 0% in single-shot for our model. However, Opus 4.8 NKIAgent achieves 74% on Level 3 (37/50, 150-task), demonstrating that the NKI-AGENT tool framework is sufficient for complex tasks when paired with stronger base models.

Common failures. Incorrect tile dimensions (partition $\neq$ 128), unsupported operation combinations, and SBUF/PSUM memory violations. Failures are *systematic*—entire categories fail uniformly—suggesting broader data coverage is more promising than RL. SFT reliably generates the load→compute→store pattern with 128-element partition but struggles with Tensor Engine matmul and multi-engine pipelining.

5.4. Example agent trace

Figure 4 shows an illustrative NKIAgent interaction on a matrix multiplication task, condensed from typical evaluation logs. This trace illustrates why tool use is essential: the agent must discover that `nl.matmul` does not exist in NKI, learn the tile shape constraints of `nisa.nc_matmul`, and fix numerical accumulation—each requiring real compiler feedback.

5.5. Limitations and future work

Our work has several limitations. First, we evaluate correctness only, not runtime performance; AccelOpt (Zhang et al., 2026) addresses this complementary axis. Second, binary reward was insufficient for GRPO; graded rewards remain future work. Third, all results are single-run without error bars; small absolute differences may not be statistically significant. Fourth, our SFT model uses only 3B active parameters; our Opus 4.8 results (77.3%) confirm that scaling model capability significantly improves results within the same framework. Finally, results are specific to Neuron SDK 2.24.

Customer Problem Statement

This research addresses **closing the software ecosystem gap for AWS custom silicon**. Amazon has deployed over 1.4M Trainium chips across three generations, yet the primary adoption barrier is software ecosystem maturity, not hardware. Trainium offers compelling economics (trn1.32xlarge $21.50/hr vs p4d.24xlarge $21.96/hr; trn2 provides 30–50% lower TCO), but customers must implement custom NKI kernels for unsupported operations.

The kernel bottleneck. NKI bypasses three of four compiler stages, requiring manual memory management and multi-engine orchestration. The ecosystem is nascent (`nki-samples`: ∼63 GitHub stars vs ∼19,100 for Triton), so LLMs have orders of magnitude less NKI training data, making automated generation particularly valuable.

Customer benefit. NKI-AGENT reduces the kernel engineering barrier for common operations, directly impacting Trainium adoption velocity. **Potential disadvantages:** generated kernels may be suboptimal vs hand-tuned implementations. **Consequences of failure:** incorrect kernels are caught by compilation or numerical verification before deployment; even the best SFT model rejects ∼75–79% of attempts, meaning the system does not produce silently wrong results. **Data bias:** training data is biased toward common PyTorch operations; mitigated by balanced curation (38% hard categories). No personal data is involved.

References

Amazon Web Services. Aws neuron sdk. https://aws.amazon.com/machine-learning/neuron/, 2024.

AWS Neuron Team. Neuron kernel interface (nki) documentation. https://awsdocs-neuron.readthedocs-hosted.com/en/latest/general/nki/, 2024a.

Table 4. **SFT data quality progression.** Each version fixes a systematic data issue. "–" = version discarded before full evaluation (issues found during spot checks).

Version	Issue Fixed	SS (60)	Δ
v1	Wrong message format	—	—
v2	64% wrong function names + overfitting	8.3%	baseline
v3	48% used 3D shapes (should be 2D)	—	—
v4	All fixed: 2D shapes, correct naming	15.0%	+81% rel.
Base (no SFT)	—	10.0%	—

Table 5. **GRPO fails to improve over SFT v4.** 200 steps of GRPO with binary reward degrades performance across all configurations.

Model	SS (60)	NKIAgent (60)	SS (150)	NKIAgent (150)
SFT v4	15.0%	**25.0%**	**19.3%**	**20.7%**
GRPO	13.3%	16.7%	15.3%	17.3%
Δ	−11%	−33%	−21%	−16%

Turn 1: Agent generates initial kernel with `nl.matmul(a_tile, b_tile)`
↪ `compile_kernel`: FAIL — `nl.matmul` does not exist; use `nisa.nc_matmul`
Turn 2: Agent switches to `nisa.nc_matmul`, incorrect tile shape (128, 512)
↪ `compile_kernel`: FAIL — free dim of lhs must equal partition dim of rhs
Turn 3: Agent transposes RHS, adds tiling loop over K dimension
↪ `compile_kernel`: OK. `verify_kernel`: FAIL — allclose atol=1e-3 failed
Turn 4: Agent adds PSUM accumulation with `nisa.nc_matmul(..., mask=...)`
↪ `compile_kernel`: OK. `verify_kernel`: PASS

Figure 4. **Illustrative NKIAgent trace** (condensed from evaluation logs). The agent iteratively discovers the correct API (`nisa.nc_matmul`), fixes tile dimension constraints, and adds proper accumulation—errors that are impossible to recover from in single-shot mode.

AWS Neuron Team. Nki samples: Reference nki kernel implementations. https://github.com/aws-neuron/nki-samples, 2024b. MIT-0 License.

DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Hu, E. J. et al. Lora: Low-rank adaptation of large language models. *ICLR*, 2022.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. Swe-bench: Can language models resolve real-world github issues? *ICLR*, 2024.

Le, H. et al. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *NeurIPS*, 2022.

Li, Y. et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

Olausson, T. X. et al. Is self-repair a silver bullet for code generation? *arXiv preprint arXiv:2306.09896*, 2024.

Ouyang, A. et al. Kernelbench: Can llms write gpu kernels? *arXiv preprint arXiv:2502.10517*, 2025.

Qwen Team. Qwen3-coder: A code-specialized mixture-of-experts language model. https://huggingface.co/Qwen/Qwen3-Coder-30B-A3B-Instruct, 2025.

Roziere, B., Lachaux, M.-A., Chanussot, L., and Lample, G. Unsupervised translation of programming languages. *NeurIPS*, 2020.

Shao, Z. et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024. Introduces GRPO (Group Relative Policy Optimization).

Shinn, N., Cassano, F., Berman, E., Gopinath, A., Narasimhan, K., and Yao, S. Reflexion: Language agents with verbal reinforcement learning. *NeurIPS*, 2023.

Shojaee, P. et al. Execution-based code generation using deep reinforcement learning. *arXiv preprint arXiv:2301.13816*, 2023.

TehraniJamsaz, A., Bhattacharjee, A., Chen, L., Ahmed, N. K., Yazdanbakhsh, A., and Jannesari, A. Coderosetta: Pushing the boundaries of unsupervised code translation for parallel programming. *NeurIPS*, 2024.

Various. Tritonbench: Benchmarking triton kernel generation, 2025.

Wen, Y., Guo, Q., Fu, Q., Li, X., Xu, J., Tang, Y., Zhao, Y., Hu, X., Du, Z., Li, L., et al. Babeltower: Learning to auto-parallelized program translation. *ICML*, 2022.

Wu, Y. et al. Cuda-agent: Agentic reinforcement learning for cuda kernel generation. *arXiv preprint arXiv:2602.24286*, 2026.

Yang, J. et al. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024.

Zhang, G., Zhu, S., Wei, A., Song, Z., Nie, A., Jia, Z., Vijaykumar, N., Wang, Y., and Olukotun, K. Accelopt: A self-improving llm agentic system for ai accelerator kernel optimization. *Proceedings of the 9th MLSys Conference*, 2026.