MODELPARTS: MODEL-GROUNDED TASKS AND ROOFLINE REWARDS FOR TRAINING KERNEL AGENTS ON EMERGING ACCELERATORS

Anonymous authors
Paper under double-blind review

ABSTRACT

As custom accelerators proliferate, kernel-optimization agents face a recurring cold start problem: the optimized libraries current methods use for training do not yet exist. Our premise is that the AI models a platform aims to train and serve can supply training data for reinforcement learning, while the accelerator supplies rewards based on its compute and memory throughput limits. We present ModelParts, which captures model computation graphs during compilation and explores alternative partitions to generate tasks whose kernel boundaries are not limited to existing libraries or manually selected motifs. To enable reinforcement learning on these tasks without baseline kernels in the target domain-specific language (DSL), we score measured latency against the hardware's roofline. We instantiate this approach with Neuron Kernel Interface (NKI), a domain-specific language for AWS Trainium and Inferentia accelerators. We construct ModelParts-6K, a corpus of 6,000 tasks generated from five models spanning distinct architectures. To isolate the effect of ModelParts data, we train models with reinforcement learning on ModelParts-6K and a synthetic dataset of PyTorch operator compositions. To evaluate transfer beyond generated tasks, we measure whether trained models can optimize the expert-written kernels in NKILib. With profiler-in-the-loop access, the ModelParts-trained policy yields a $1.11\times$ aggregate speedup on NKILib, surpassing both frontier baselines without profiler access.

1 INTRODUCTION

Training and inference can approach an accelerator's performance limits only through efficient kernels, yet developing them requires hardware-specific expertise in tiling, data movement, and memory layout. Even mature ecosystems must redesign kernels for new architecture releases: FlashAttention-3 was tuned for Hopper H100 (Shah et al., 2024), but FlashAttention-4, its B200 redesign, arrived more than a year after Blackwell entered production (NVIDIA, 2024; Zadouri et al., 2026). Benchmarks, coding agents, and reinforcement learning systems have begun to automate this work for CUDA and Triton (Ouyang et al., 2025; Baronio et al., 2026; Dai et al., 2026; Woo et al., 2025), as well as NKI, the kernel language for AWS Trainium and Inferentia (Zhang et al., 2026; Tang et al., 2026b).

Current training pipelines depend on two resources that emerging platforms initially lack: realistic tasks and reliable performance rewards. Existing task collections curate operators and model fragments (Ouyang et al., 2025), synthesize compositions of sampled PyTorch operations (Dai et al., 2026), or mine code from repositories (Paliskara & Saroufim, 2025). These approaches respectively impose human-selected boundaries, construct cross-operation structure synthetically rather than inherit it from model graphs, or depend on an existing code ecosystem. Performance rewards are typically measured relative to a baseline kernel or compiled PyTorch reference (Baronio et al., 2026; Dai et al., 2026; Li et al., 2026). Their scale therefore depends on baseline quality and is least reliable when expert implementations are scarce. Bootstrapping an emerging platform requires tasks and performance supervision that do not presuppose an existing kernel ecosystem.

Even a new platform provides two sources for these missing ingredients: the models it is built to train and serve, and the hardware itself. Compiling these models exposes operation graphs from

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.

which executable references can be extracted. Dividing an operation graph into disjoint parts defines candidate kernels, and exploring alternative partitions yields new task boundaries with executable correctness references. Hardware profiling measures each candidate's latency, while peak compute throughput and memory bandwidth define a roofline lower bound (Williams et al., 2009) against which that latency can be scored without a baseline implementation. The models can therefore supply the tasks and the hardware can supply the supervision.

We instantiate these ideas as ModelParts and evaluate whether reinforcement learning on its generated NKI tasks transfers to held-out generated tasks and expert-written NKILib kernels. ModelParts training transfers more broadly across generated-task sources than training on synthetic operator compositions. Combined with PITL, it achieves the strongest Qwen3.8-27B performance on both held-out generated tasks and NKILib, reaching 59.9 overall Pass@8 and $1.11\times$ aggregate speedup, respectively. These results support the premise that kernel agents can begin training before a mature kernel ecosystem exists.

Our contributions are:

- **ModelParts**, a task-generation method that explores alternative partitions of compiled model graphs to produce tasks with executable correctness references. We instantiate it as **ModelParts-6K**, a corpus of 6,000 NKI tasks derived from five architecturally distinct model components across three execution settings.
- **A correctness-gated roofline reward** that scores correct kernels by their measured latency relative to a hardware-derived lower bound, providing performance supervision without a baseline implementation in the target language.
- **A controlled comparison** of model-grounded and synthetic operator-compositions training tasks under identical training settings, with and without profiler-in-the-loop access, including transfer to expert-written NKILib kernels.

2 Related Work

Language models for kernel generation and optimization. Yu et al. (2026) survey the rapidly growing use of language models to generate and optimize accelerator kernels. Benchmarks such as KernelBench, TritonBench, MultiKernelBench, and TritonGym evaluate kernel generation across languages, accelerator platforms, and agentic workflows (Ouyang et al., 2025; Li et al., 2025a; Wen et al., 2025; Guan et al., 2026). Agentic systems iteratively generate, execute, and refine kernels using multi-agent workflows, hardware feedback, bandit-guided selection, or evolutionary search (Wei et al., 2025; Lei et al., 2025; Dong et al., 2026; Tschand et al., 2025; Zhang et al., 2025; Ran et al., 2026; Andrews & Witteveen, 2025; Wiedemann et al., 2026); our profiler-in-the-loop condition exposes similar execution feedback. Kevin, CUDA Agent, ConCuR, AutoTriton, and Dr. Kernel train kernel-specialized policies with supervised or reinforcement learning (Baronio et al., 2026; Dai et al., 2026; Kong et al., 2025; Li et al., 2025b; Liu et al., 2026).

Kernel task construction. Existing task collections draw on curated benchmarks, mined code, and synthetic programs. KernelBench curates individual operators, operator sequences, and full architectures (Ouyang et al., 2025). KernelBook pairs PyTorch modules mined from GitHub with Triton kernels produced by `torch.compile` (Paliskara & Saroufim, 2025), and subsequent training datasets build on it (Fisches et al., 2025; Woo et al., 2025). CUDA-Agent-Ops composes sampled PyTorch operators into synthetic programs (Dai et al., 2026). Like KernelBook, ModelParts uses the compiler, but it samples partitions beyond the one the compiler selects.

Compiler fusion and graph partitioning. Compilers such as XLA, TVM, Ansor, and TorchInductor decide which operations to fuse, typically with rules or heuristics, and search over scheduling and tiling choices (OpenXLA, 2017; Chen et al., 2018; Zheng et al., 2020; Ansel et al., 2024). Recent work uses language models to guide compiler optimization decisions (Tang et al., 2026a; Pan et al., 2025). These systems choose partitions to produce one efficient program. ModelParts instead samples many partitions as training tasks, and its latency estimate serves only to bias sampling toward lower latency schedules.

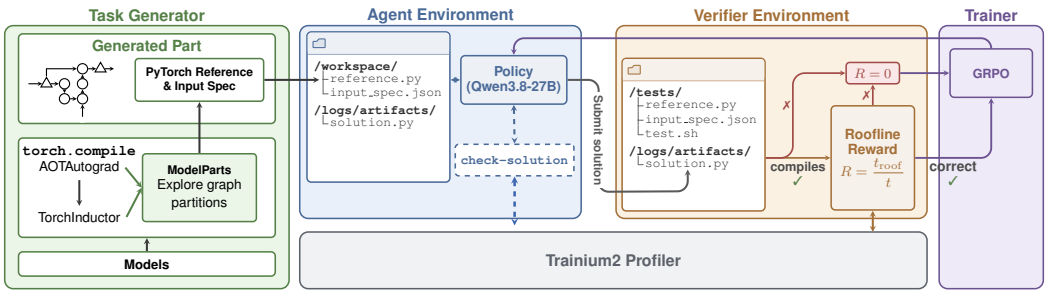


Figure 1: **End-to-end training flow.** Each sampled graph part yields a PyTorch reference and input specification (left). The policy develops an equivalent NKI kernel and may query PITL (center). Final submissions receive $R = 0$ on compilation or hidden-input correctness failure and $R = t_{\text{roof}}/t$ otherwise; GRPO uses this reward to update the policy (right).

Rewards and verification. Existing reinforcement learning methods for kernel generation commonly derive rewards from speedup relative to a baseline kernel or compiled PyTorch reference (Baronio et al., 2026; Dai et al., 2026; Li et al., 2026). This makes the reward depend on baseline quality rather than candidate efficiency alone: weak baselines inflate apparent gains. Unbounded speedup ratios can also produce outliers and bias training toward tasks with weak baselines (Dai et al., 2026). The roofline ratio, by contrast, is bounded near one by the hardware limit (Williams et al., 2009).

Emerging accelerator DSLs. Kernel generation for accelerators beyond NVIDIA GPUs has received less attention. AscendKernelGen targets Huawei Ascend NPUs (Cao et al., 2026), and MultiKernelBench evaluates generation across several platforms (Wen et al., 2025). For NKI, AccelOpt optimizes kernels at test time through search and an accumulated optimization memory, without training the model, and scores kernels against peak hardware throughput on NKIBench, a suite extracted from LLM workloads (Zhang et al., 2026). NKI-Agent recreates the CUDA-Agent-Ops task-sampling procedure for NKI and trains with GRPO using only a binary correctness reward, which fails to improve over supervised fine-tuning (Tang et al., 2026b). EvoKernel similarly addresses the NPU cold start at test time through memory-based retrieval (Zheng et al., 2026). We instead train on generated, model-grounded tasks with a reward that credits performance as well as correctness.

3 METHODS

Our approach consists of three components: (1) a task-generation pipeline that samples alternative kernel partitions from model operation graphs, (2) an NKI kernel agent loop operating in an isolated environment with optional PITL access, and (3) a verifier that produces a roofline-based reward used to train the policy with GRPO. (Figure 1).

3.1 MODELPARTS: TASK GENERATION FROM MODEL-GRAPH PARTITIONS

To generate kernel-writing tasks at scale, we capture operation graphs during model compilation and explore ways of partitioning each graph into candidate kernels. During `torch.compile`, AOTAutograd lowers a PyTorch model into a directed acyclic graph (DAG) of ATen operations, which TorchInductor then lowers into a schedule of executable kernels (Ansel et al., 2024). The schedule induces a partition of the operation DAG. ModelParts starts from this compiler-derived partition and walks over nearby partitions that preserve the computation graph while changing its boundaries (Figure 2).

Inducing a kernel partition of the operation DAG. To construct the initial partition P_0, ModelParts maps each scheduled kernel to its provenance set of operations in the captured ATen DAG. These sets need not form a partition: they can overlap when one ATen operation lowers to multiple nodes in the compiler's intermediate representation that are scheduled into different kernels, and they can omit operations absorbed into indexing logic, such as views and transposes. ModelParts

converts them into disjoint parts by processing the scheduled kernels in the compiler's topological order. An operation appearing in multiple provenance sets is assigned to the earliest corresponding kernel. An operation omitted from every provenance set is assigned to the part containing its first downstream consumer. Parts belonging to a cycle in the resulting quotient graph are merged until the graph is acyclic. The resulting parts constitute P_0, the compiler-induced partition of the operation DAG.

Exploring model-grounded kernels. The compiler-derived partition P_0 provides an executable decomposition of the captured model graph, but its boundaries are constrained by implemented lowerings, fusion rules, kernel templates, and auto-tuning choices rather than hardware efficiency alone. Each sampled part defines a kernel-writing task with an executable PyTorch correctness reference, while direct hardware profiling measures the performance of generated implementations. This makes alternative kernel boundaries usable as training tasks without requiring a pre-existing kernel in the target language. To sample such alternatives at scale, we perform a random walk $(P_i)_{i=0}^{n}$ over partitions of the same fixed operator DAG. From the current partition P, each step first selects a move type μ uniformly from:

- **SHIFT**: moves one operation across a boundary into an adjacent part,
- **FUSE**: removes a boundary by merging two adjacent parts, amd
- **SPLIT**: introduces a boundary by dividing one part into a prefix and suffix of a topological ordering of its operations,

then selects uniformly among the $N_\mu(P)$ available moves of that type, yielding a proposal P'. SHIFT and SPLIT each produce two changed parts, whereas FUSE produces one.

For a candidate kernel k evaluated on input x, define its roofline lower bound on latency as

$$t_{\mathrm{roof}}(k;x) = \max\left(\frac{c_k(x)}{r_c}, \frac{m_k(x)}{r_m}\right), \tag{1}$$

where $c_k(x)$ is the required FLOP count, $m_k(x)$ is the compulsory traffic across the accelerator memory interface, r_c is peak compute throughput, and r_m is peak memory bandwidth. Let $x_k = x_k(P,x)$ denote the boundary inputs received by part k when partition P executes on capture input x. We estimate partition latency with a model that assumes kernels execute serially:

$$T(P;x) = |P| t_{\mathrm{launch}} + \sum_{k\in P} t_{\mathrm{roof}}(k;x_k),$$

where t_{launch} is a fixed per-kernel launch latency. Because each walk uses a fixed capture input, we suppress x below and write $T(P)$. We use T as the energy function of a Metropolis–Hastings chain that favors lower-latency partitions. For the current partition P and a distinct valid proposal P', the transition probability is

$$\Pr(P \to P') = q(P' \mid P)\min\left\{1, \exp\left(-\frac{T(P')-T(P)}{\lambda}\right)\frac{q(P \mid P')}{q(P' \mid P)}\right\},$$

where the forward and reverse proposal probabilities are

$$q(P' \mid P) = \frac{1}{3N_\mu(P)}, \qquad q(P \mid P') = \frac{1}{3N_{\mu^{-1}}(P')}.$$

Here, μ^{-1} is the move type that reverses the selected move, and λ is the temperature, which controls how often the walk accepts higher-latency partitions. We calibrate λ from the one-move neighborhood of P_0 and a discarded burn-in chain to target a fixed acceptance rate for higher-latency proposals.

The partition latency estimate discourages fragmentation through the launch term and memory traffic at kernel boundaries. It does not, however, capture practical limits on a candidate kernel's working set size. We therefore restrict valid partitions using bounds derived from P_0. A part may not exceed the maximum peak live bytes among parts in P_0, and its operation count, dependency depth, peak live-value count, and input/output arity may not exceed $1.5\times$ their corresponding maxima in P_0.

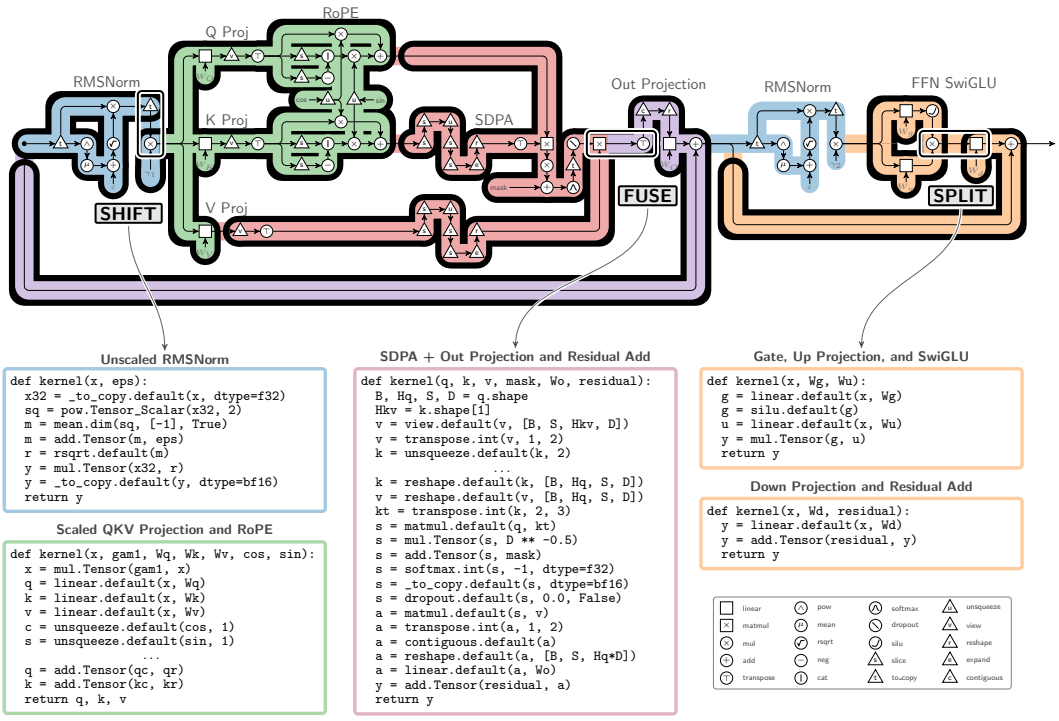


Figure 2: **ModelParts partition-walk moves.** Colored tubes depict the compiler-derived partition of a representative decoder layer's post-AOTAutograd ATen operation graph. Solid black borders outline new kernels generated by shift, fuse, and split moves. The executable PyTorch code corresponding to each generated part serves as its correctness reference.

Each changed part k defines a candidate task $\tau_k = (F_k, s_k)$, where F_k is its executable PyTorch reference and s_k records the input shapes and dtypes observed at its boundary during capture. Each task contains one captured shape-and-dtype specification; the verifier samples three hidden input values with that same specification rather than testing additional shapes. Because models frequently reuse the same operation subgraphs, we compute a canonical Merkle hash (Appendix A.4) of each part's operation DAG and render only one task per distinct hash.

3.2 Agent Architecture

Agent loop. We use Harbor (Harbor Framework Team, 2026) to run each task in an isolated Docker container provisioned with the executable PyTorch reference and a pinned Neuron SDK providing the NKI Python package. Mini-SWE-Agent drives a multi-turn interaction using custom system and instance prompts (Listings 1 and 2). At each turn, the policy emits one Bash command and receives its output as the next observation. The policy writes and iterates on its solution until marking it complete or reaching the 64-step or 131,072-token limit.

Profiler-in-the-loop access. PITL adds a single agent-callable command, `check-solution`, that compiles the current NKI implementation for Trainium and submits the resulting artifact to a remote hardware verifier. The command returns compiler diagnostics or, after successful compilation, correctness and latency results, allowing the agent to revise and retry. Without PITL, `check-solution` is unavailable. Only the final submitted solution is verified on Trainium. Listing 3 shows the PITL-specific prompt changes.

3.3 Correctness-Gated Roofline Reward

Baseline-free performance supervision. Existing reinforcement learning methods for kernel generation use speedup over a baseline kernel or PyTorch implementation as the reward (Wang et al.,

2026; Baronio et al., 2026; Li et al., 2026). However, speedup depends on baseline quality: weak baselines inflate apparent gains, while tasks without a baseline cannot be scored. We instead measure performance against a hardware bound.

For each task $\tau_k = (F_k, s_k)$, a submitted solution f is first compiled. If compilation succeeds, the resulting artifact is sent to the hardware verifier, which generates three hidden inputs $\mathcal{X}_k = \{x_1, x_2, x_3\}$ from s_k. For each x, the verifier executes $F_k(x)$ under `FlopCounterMode` to obtain the expected outputs and compute the FLOP count $c_k(x)$. It computes compulsory traffic $m_k(x)$ as one read of each distinct byte in the input tensors x and one write of each distinct byte in the reference outputs $F_k(x)$, merging overlapping views that share storage. Together with the hardware rates r_c and r_m for the verifier's Trainium2 logical NeuronCore configuration (LNC $= 2$; Appendix A.5), these quantities determine $t_{\text{roof}}(k; x)$ in Equation 1.

The verifier compares $f(x)$ with $F_k(x)$ for every $x \in \mathcal{X}_k$ according to Appendix A.8. A compilation failure or mismatch on any input yields zero reward. To reduce cold-start effects and measurement noise, each correct artifact is warmed up once and then timed three times per input. Let $t_j(f; x)$ denote its j-th measured latency on x. The reward is

$$R(f; \tau_k) = \underset{x \in \mathcal{X}_k}{\text{median}} \; \underset{j \in \{1,2,3\}}{\text{median}} \; \frac{t_{\text{roof}}(k; x)}{t_j(f; x)}. \tag{2}$$

Thus, for correct solutions, the reward measures how closely the candidate's measured latency approaches the theoretical minimum predicted by the roofline model.

3.4 Policy Optimization

We assign the final verifier reward to the submitted trajectory and optimize the policy with GRPO. Trajectories generated for the same task form a reward group, and their group-relative advantages are used in the clipped policy objective.

4 Experiments

4.1 Task Corpora

ModelParts-6K. To instantiate ModelParts, we select one layer-level component from five models spanning architectural features such as sliding-window attention, multi-head latent attention, mixture-of-experts, and loop transformers. For each model, we run an independent partition walk in each of three execution settings: prefill forward, prefill backward, and autoregressive decode forward, yielding 15 walks summarized in Table 1. Appendix A.1 details the capture setup, partition walks, and corpus assembly. We set λ to the largest temperature for which the estimated acceptance probability of uphill proposals is at most $1/4$. We exclude tasks corresponding to parts of P_0, deduplicate within and across walks, and retain 400 distinct tasks per walk. Splitting each walk into 360 training and 40 evaluation tasks yields ModelParts-6K, comprising 5,400 training and 600 evaluation tasks. Appendix A.10.1 shows a complete task instance. We plan to release ModelParts-6K publicly.

NKI-Agent-Ops-6K. We adapt CUDA-Agent-Ops-6K (Dai et al., 2026), a collection of 6,000 synthetic PyTorch operation chains, to NKI. Each PyTorch reference defines a `Model` with constructor and runtime inputs, while its parameters and buffers are held as module state. We instantiate the model and functionalize it using `torch.export`, to produce a class-free ATen reference with explicit runtime and model-state inputs. Constructor values needed to reconstruct the computation are retained in the generated reference. Each resulting task provides this executable reference and concrete input specifications and asks the policy to implement the same function as an NKI kernel. Appendix A.10.2 shows an original class-based CUDA-Agent-Ops program and the task produced by this adaptation.

NKILib. To study whether learned optimization skill transfers to production code, we evaluate ten expert-written kernels from NKILib spanning attention, MLP, output projection, QKV projection, mixture-of-experts, and rotary positional embeddings. Agents may inspect the installed implementation, so the benchmark measures source-visible optimization rather than kernel writing from

Table 1: **ModelParts-6K partition-walk statistics.** Across prefill forward (PF), decode forward (DF), and prefill backward (PB), walks generate 2.8 to 31.7 tasks per 100 steps containing 4–10 operations per reference ($|k|$) from initial partitions with 15–31 parts ($|P_0|$). Yields below 25 highlight the importance of content-based deduplication.

	$\lvert P_0\rvert$			Median $\lvert k\rvert$			Mem-bound (%)			Task yield		
Model	PF	DF	PB	PF	DF	PB	PF	DF	PB	PF	DF	PB
Mistral 7B	18	18	24	4	5	7	33	100	33	4.4	14.9	21.5
Mixtral 8x7B	16	17	30	7	10	8	63	100	41	7.3	23.9	31.7
Qwen3 8B	20	21	31	5	6	6	38	100	60	10.9	20.3	22.4
Ouro 1.4B	15	17	29	6	7	5	32	100	39	15.0	25.7	23.1
DeepSeek V2 Lite	15	16	27	5	5	6	31	100	35	2.8	10.2	29.2

scratch. Only one representative input specification is visible; final verification checks correctness and measures latency over a range of three input sizes. Appendix A.10.3 shows a complete QKV optimization task.

Training conditions. We use a 2×2 factorial design to isolate the effects of training data and PITL: $\{\text{ModelParts-6K}, \text{NKI-Agent-Ops-6K}\} \times \{\text{without PITL}, \text{with PITL}\}$. All remaining training settings are held fixed across conditions.

Models and optimization. We post-train Qwen3.8-27B from its base checkpoint using GRPO. Rollouts are sampled at temperature 1.0. We make one pass over each training corpus with a group size of 8. We use a learning rate of 2.5×10^{-7}, limit each response at 16,384 tokens, and limit the complete agent trajectory at 131,072 tokens. Training runs asynchronously on separate rollout and training nodes and uses truncated importance sampling (TIS) to correct train–rollout mismatch (Yao et al., 2025).

4.2 Evaluation Setup

Baselines. We evaluate the base and post-trained Qwen3.8-27B policies. Opus 5 provides a frontier-model baseline on the generated tasks and NKILib, while GPT-6 Sol provides an additional frontier baseline on NKILib. For NKILib evaluations only, frontier models receive a maximum of 128 agent steps, 32,768 output tokens, and 262,144 context tokens.

Metrics. We report best-of-eight performance. On generated tasks, Pass@8 is the fraction of completed eight-trial groups containing at least one correct generated kernel, and Reward@8 averages the maximum roofline reward across groups. For each NKILib task, we retain the best correct kernel produced at any step of any of the eight rollouts. Its score is the geometric-mean speedup over the expert implementation across the three test inputs or varied dimensions, and the aggregate score is the geometric mean across tasks. Figure 7 in Appendix A.7 reconstructs the verified candidate trajectories for the rollouts that produced these selected kernels.

5 Results

5.1 Generated-Task Performance

Table 2 compares performance on held-out ModelParts-6K and NKI-Agent-Ops-6K tasks across model, training-corpus, and profiler-access conditions.

Reinforcement learning improves correctness and efficiency. Without PITL, ModelParts and NKI-Agent-Ops training raise Qwen3.8-27B's overall Pass@8 from 29.2 to 49.7 and 44.9, respectively, and Reward@8 from 0.095 to 0.215 and 0.183. Across PITL settings, every trained Qwen condition exceeds Opus 5 in overall Reward@8. Base Qwen also exceeds Opus 5 on ModelParts Pass@8 (34.2 versus 21.5), while Opus 5 performs consistently worse on ModelParts than on NKI-

Table 2: **Best-of-eight performance on held-out generated tasks.** Pass@8 is the percentage of completed eight-rollout groups containing a correct implementation, and Reward@8 averages the maximum roofline reward across groups. Overall pools ModelParts and NKI-Agent-Ops. Checkmarks denote PITL access or reinforcement learning on ModelParts (MP) or NKI-Agent-Ops (NAO).

		RL		ModelParts Eval		NKI-Agent-Ops Eval		Overall	
Model	**PITL**	**MP**	**NAO**	**Pass@8**	**Reward@8**	**Pass@8**	**Reward@8**	**Pass@8**	**Reward@8**
Opus 5				21.5	0.046	29.0	0.101	25.3	0.073
Opus 5	✓			32.5	0.077	43.1	0.140	37.8	0.108
Qwen3.8-27B				34.2	0.095	23.2	0.096	29.2	0.095
Qwen3.8-27B	✓			38.3	0.063	37.1	0.061	37.7	0.062
Qwen3.8-27B		✓		67.2	0.253	32.2	0.177	49.7	0.215
Qwen3.8-27B	✓	✓		84.9	0.293	34.8	0.173	**59.9**	**0.233**
Qwen3.8-27B			✓	52.8	0.206	32.9	0.159	44.9	0.183
Qwen3.8-27B	✓		✓	56.8	0.241	50.9	0.223	53.9	0.232

Agent-Ops, indicating that model-grounded parts are the harder held-out distribution for an untrained frontier model.

ModelParts training transfers more broadly. Without PITL, the ModelParts-trained policy nearly matches the NKI-Agent-Ops-trained policy on NKI-Agent-Ops Pass@8 (32.2 versus 32.9) and exceeds it on Reward@8 (0.177 versus 0.159). The reverse transfer is weaker: on ModelParts, NKI-Agent-Ops training reaches 52.8 Pass@8 and 0.206 Reward@8, compared with 67.2 and 0.253 after ModelParts training. Model-grounded tasks therefore cover much of what synthetic compositions teach.

Profiler access interacts with training. PITL raises base Qwen's overall Pass@8 from 29.2 to 37.7 but lowers Reward@8 from 0.095 to 0.062, suggesting that feedback primarily helps the untrained policy repair correctness. After training, PITL improves both metrics. ModelParts training with PITL achieves the best overall Pass@8 (59.9) and ModelParts results (84.9 Pass@8 and 0.293 Reward@8), but changes NKI-Agent-Ops Pass@8 only from 32.2 to 34.8. By contrast, PITL raises the NKI-Agent-Ops-trained policy from 32.9 to 50.9 on its own distribution. The two PITL-trained policies nevertheless reach similar overall Reward@8 (0.233 and 0.232). Although the PITL-trained policies have similar overall Reward@8 (0.233 versus 0.232), their conditional distributions differ: correct ModelParts-trained solutions achieve higher roofline rewards (Figure 6; Appendix A.6).

5.2 Transfer to Expert-Written Kernels

Expert kernels leave limited headroom. Table 3 reports best-of-eight speedup over ten expert-written NKILib implementations. Because each task begins with a correct expert kernel, scores cluster near $1.00\times$; values below $1.00\times$ indicate that the best correct candidate was slower. MLP, output-projection CTE, and QKV remain near this floor across conditions. Opus 5 with PITL achieves the strongest aggregate result ($1.27\times$), while GPT-6 Sol reaches $1.00\times$ without PITL and $1.07\times$ with PITL.

PITL improves transfer. PITL improves every matched configuration: Opus 5 rises from $1.06\times$ to $1.27\times$, GPT-6 Sol from $1.00\times$ to $1.07\times$, base Qwen from $1.03\times$ to $1.06\times$, ModelParts-trained Qwen from $1.05\times$ to $1.11\times$, and NKI-Agent-Ops-trained Qwen from $1.02\times$ to $1.06\times$. ModelParts with PITL is the strongest Qwen condition and surpasses both frontier models without PITL. Its gains concentrate in attention (1.13–$1.24\times$) and MoE TKG ($1.43\times$); the latter remains $1.31\times$ without PITL, indicating transfer from model-grounded training. On MoE TKG, Opus 5 with PITL streamlines memory movement and accumulation within one kernel ($1.59\times$), whereas ModelParts-trained Qwen with PITL partitions the work across two cores and exchanges partial results ($1.43\times$).

Table 3: **Best-of-eight speedup over expert NKILib implementations.** TKG and CTE denote token generation and context encoding. Bold indicates the best-in-class speedup exceeding 1%.

		RL		Attention			MLP	Out Proj		QKV	MoE		RoPE	Agg
Model	**PITL**	**MP**	**NAO**	**TKG**	**CTE**	**Bwd**		**CTE**	**TKG**		**TKG**	**CTE**		
Opus 5				1.06	1.06	1.22	1.00	1.00	0.95	0.99	1.23	**1.04**	1.96	1.06
Opus 5	✓			**1.24**	**1.61**	**1.27**	1.00	1.00	**1.18**	0.98	**1.59**	1.03	**2.25**	**1.27**
GPT-6 Sol				0.99	1.00	1.00	1.01	1.00	1.01	0.99	1.04	1.00	1.00	1.00
GPT-6 Sol	✓			1.09	1.15	1.10	**1.03**	1.01	1.02	**1.02**	1.08	1.00	1.21	1.07
Qwen3.8-27B				1.06	1.06	1.00	1.00	1.01	1.00	0.99	1.03	1.00	1.11	1.03
Qwen3.8-27B	✓			1.03	1.06	**1.14**	1.00	1.00	1.03	0.98	1.02	**1.07**	1.27	1.06
Qwen3.8-27B		✓		1.03	1.06	1.00	1.00	1.00	1.00	1.00	1.31	1.06	0.97	1.05
Qwen3.8-27B	✓	✓		**1.24**	**1.13**	**1.14**	0.99	1.01	**1.04**	1.00	**1.43**	**1.07**	1.15	**1.11**
Qwen3.8-27B			✓	1.05	1.06	1.07	1.00	1.00	1.00	1.00	1.02	**1.07**	0.96	1.02
Qwen3.8-27B	✓		✓	1.06	1.06	1.07	1.00	1.01	**1.04**	1.00	1.02	**1.07**	**1.29**	1.06

Frontier optimization reveals further headroom. A gap to Opus 5 with PITL remains on attention CTE ($1.13\times$ versus $1.61\times$) and RoPE ($1.15\times$ versus $2.25\times$). On RoPE, Opus fills all 128 partition lanes, shards work for contiguous transfers, and reduces vector and DMA overhead, demonstrating optimization headroom beyond the expert implementation.

6 Limitations

ModelParts is limited by the abstraction at which it recovers and explores kernels. It treats captured ATen operations as indivisible, although TorchInductor may lower one operation into multiple intermediate-representation nodes. Consequently, the walk cannot represent intra-operation boundaries such as tensor-axis sharding. Moreover, P_0 comes from one TorchInductor configuration targeting a different accelerator than Trainium. The walk objective and reward also use idealized roofline models. In particular, the verifier assumes a fixed throughput profile, omitting engine-specific throughput, on-chip capacity, and execution overheads. ModelParts-6K contains fixed-shape tasks drawn from five decoder components, and each generated task is evaluated only at its captured input specification. Training therefore does not directly reward shape-polymorphic implementations or robustness across input dimensions. Its retained tasks contain at most 17 operations, and larger NKILib optimization tasks differ materially from the training distribution. Because Trainium already has an SDK and expert kernels, the experiment emulates a platform cold start by withholding target-language baselines rather than observing a platform at launch.

Future work could extend ModelParts to larger fused kernels, more layers, and multimodal architectures. Applying the pipeline to a newly released accelerator would test the cold-start premise directly. Combining model-grounded and synthetic corpora may further improve transfer, given their complementary strengths under PITL.

7 Conclusion

We showed that the models a platform must serve and the hardware it runs on can together bootstrap a kernel-optimization agent without baseline kernels in the target language. ModelParts turns compiled model graphs into executable kernel-writing tasks by sampling alternative partitions, and a correctness-gated roofline reward scores them against hardware limits rather than existing implementations. On NKI, reinforcement learning on ModelParts-6K raises Qwen3.8-27B's overall Pass@8 from 29.2 to 49.7 and transfers to synthetic operator compositions. With profiler feedback, the ModelParts-trained policy achieves a $1.11\times$ aggregate speedup over expert-written NKILib kernels, exceeding both frontier models without profiler access. A gap to the strongest frontier configuration remains. These results suggest that kernel agents for emerging accelerators can begin training before a mature kernel ecosystem exists.

AI USE STATEMENT

Language models appear in this work in two distinct roles. As *research subjects*, Qwen3.8-27B is the policy we train, and Opus 5 and GPT-6 Sol serve as evaluated baselines; their use is described in Sections 4 and 5. This statement concerns the separate use of generative AI tools as *research assistants*.

Required disclosures. We used generative AI tools to assist in implementing parts of the ModelParts pipeline, analyze experimental results. We did not use AI to generate synthetic data sets, help develop theoretical models or conceptual frameworks, formulate mathematical claims, provide critical ingredients for proving mathematical claims, assist in the writing of proofs, propose or refine hypotheses, design or provide feedback on research methodology or experiments, assist with translation, clean and reformat dataset, support qualitative and thematic data analysis, or interpret results.

REFERENCES

Amazon Web Services. AWS Trainium2 Architecture. AWS Neuron Documentation, 2026. URL https://awsdocs-neuron.readthedocs-hosted.com/en/latest/about-neuron/arch/neuron-hardware/trainium2.html. Accessed September 24, 2026.

Martin Andrews and Sam Witteveen. Gpu kernel scientist: An llm-driven framework for iterative kernel optimization. *arXiv preprint arXiv:2506.20807*, 2025.

Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. PyTorch 2: Faster machine learning through dynamic Python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024. doi: 10.1145/3620665.3640366.

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 (ICLR)*, 2026. arXiv:2507.11948.

Xinzi Cao, Jianyang Zhai, Pengfei Li, Zhiheng Hu, Cen Yan, Bingxu Mu, Guanghuan Fang, Bin She, Jiayu Li, Yihan Su, Dongyang Tao, Xiansong Huang, Fan Xu, Feidiao Yang, Yao Lu, Chang-Dong Wang, Yutong Lu, Weicheng Xue, Bin Zhou, and Yonghong Tian. AscendKernelGen: LLM-driven kernel generation for NPUs. In *Findings of the Association for Computational Linguistics: ACL 2026*, 2026. arXiv:2601.07160.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 578–594, 2018.

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.

Juncheng Dong, Yang Yang, Tao Liu, Yang Wang, Feng Qi, Vahid Tarokh, Kaushik Rangadurai, and Shuang Yang. STARK: Strategic team of agents for refining kernels. In *International Conference on Learning Representations (ICLR)*, 2026. arXiv:2510.16996.

Zacharias Fisches, Sahan Paliskara, Simon Guo, Alex Zhang, Joe Spisak, Chris Cummins, Hugh Leather, Joe Isaacson, Aram Markosyan, and Mark Saroufim. KernelLLM. https://huggingface.co/facebook/KernelLLM, 2025.

Yue Guan, Huihui Qi, Wanghui Ding, Yuedong Yang, Fangqing Liu, and Xin Jin. Tritongym: A benchmark for agentic llm workflows in triton gpu code generation. In *International Conference on Machine Learning (ICML)*, 2026. https://openreview.net/forum?id=Htq2loTQTL.

Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments. Zenodo, 2026. https://doi.org/10.5281/zenodo.20953922.

Lingcheng Kong, Jiateng Wei, Hanzhang Shen, and Huan Wang. Concur: Conciseness makes state-of-the-art kernel generation. *arXiv preprint arXiv:2510.07356*, 2025.

Kelun Lei, Hailong Yang, Huaitao Zhang, Xin You, Kaige Zhang, Zhongzhi Luan, Yi Liu, and Depei Qian. Pragma: A profiling-reasoned multi-agent framework for automatic kernel optimization. *arXiv preprint arXiv:2511.06345*, 2025.

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, 2025a. doi: 10.18653/v1/2025.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.

Xiaoya Li, Albert Wang, Guoyin Wang, Jiwei Li, and Chris Shum. CUDA-L1: Improving CUDA optimization via contrastive reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2026. arXiv:2507.14111.

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 *International Conference on Machine Learning (ICML)*, 2026. arXiv:2602.05885.

NVIDIA. NVIDIA announces financial results for third quarter fiscal 2025. https://nvidianews.nvidia.com/news/nvidia-announces-financial-results-for-third-quarter-fiscal-2025, November 2024. Press release, November 20, 2024.

OpenXLA. XLA: Optimizing compiler for machine learning. https://openxla.org/xla, 2017.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? In *International Conference on Machine Learning (ICML)*, 2025. arXiv:2502.10517.

Sahan Paliskara and Mark Saroufim. KernelBook. Hugging Face dataset, 2025. https://huggingface.co/datasets/GPUMODE/KernelBook.

Haolin Pan, Hongyu Lin, Haoran Luo, Yang Liu, Kaichun Yao, Libo Zhang, Mingjie Xing, and Yanjun Wu. Compiler-R1: Towards agentic compiler auto-tuning with reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. arXiv:2506.15701.

Dezhi Ran, Shuxiao Xie, Mingfang Ji, Anmin Liu, Mengzhou Wu, Yuan Cao, Yuzhe Guo, Hao Yu, Linyi Li, Yitao Hu, Wei Yang, and Tao Xie. KernelBand: Steering LLM-based kernel optimization via hardware-aware multi-armed bandits. In *International Conference on Machine Learning (ICML)*, 2026. arXiv:2511.18868.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. FlashAttention-3: Fast and accurate attention with asynchrony and low-precision. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Annabelle Sujun Tang, Christopher Priebe, Rohan Mahapatra, Lianhui Qin, and Hadi Esmaeilzadeh. Reasoning compiler: Llm-guided optimizations for efficient model serving. *Advances in Neural Information Processing Systems*, 38:106904–106930, 2026a.

Junjie Tang, Jun Huan, Hao Zhou, Yuhao Zhang, and Lin Wang. NKI-Agent: Domain-specific fine-tuning and agentic tool use for neuron kernel generation. *arXiv preprint arXiv:2607.04395*, 2026b.

Arya Tschand, Muhammad Awad, Ryan Swann, Kesavan Ramakrishnan, Jeffrey Ma, Keith Lowery, Ganesh Dasika, and Vijay Janapa Reddi. Swizzleperf: Hardware-aware llms for gpu kernel performance optimization. *arXiv preprint arXiv:2508.20258*, 2025.

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.

Anjiang Wei, Tianran Sun, Yogesh Seenichamy, Hang Song, Anne Ouyang, Azalia Mirhoseini, Ke Wang, and Alex Aiken. Astra: A multi-agent system for gpu kernel performance optimization. *arXiv preprint arXiv:2509.07506*, 2025.

Zhongzhen Wen, Yinghui Zhang, Zhong Li, Zhongxin Liu, Linna Xie, and Tian Zhang. Multikernelbench: A multi-platform benchmark for kernel generation. *arXiv preprint arXiv:2507.17773*, 2025.

Nina Wiedemann, Quentin Leboutet, Michael Paulitsch, Diana Wofk, and Benjamin Ummenhofer. Kernelfoundry: Hardware-aware evolutionary gpu kernel optimization. In *International Conference on Machine Learning (ICML)*, 2026. arXiv:2603.12440.

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009. doi: 10.1145/1498765.1498785.

Jiin Woo, Shaowei Zhu, Allen Nie, Zhen Jia, Yida Wang, and Youngsuk Park. TritonRL: Training LLMs to think and code Triton without cheating. *arXiv preprint arXiv:2510.17891*, 2025.

Feng Yao, Liyuan Liu, Dinghuai Zhang, Chengyu Dong, Jingbo Shang, and Jianfeng Gao. On the rollout-training mismatch in modern rl systems. In *NeurIPS Workshop on Efficient Reasoning*, 2025.

Yang Yu, Peiyu Zang, Chi Hsu Tsai, Haiming Wu, Yixin Shen, Jialing Zhang, Haoyu Wang, Zhiyou Xiao, Jingze Shi, Yuyu Luo, Wentao Zhang, Chunlei Men, Guang Liu, and Yonghua Lin. Towards automated kernel generation in the era of LLMs. *arXiv preprint arXiv:2601.15727*, 2026.

Ted Zadouri, Markus Hoehnerbach, Jay Shah, Timmy Liu, Vijay Thakkar, and Tri Dao. FlashAttention-4: Algorithm and kernel pipelining co-design for asymmetric hardware scaling. *arXiv preprint arXiv:2603.05451*, 2026.

Genghan Zhang, Shaowei Zhu, Anjiang Wei, Zhenyu Song, Allen Nie, Zhen Jia, Nandita Vijaykumar, Yida Wang, and Kunle Olukotun. AccelOpt: A self-improving LLM agentic system for AI accelerator kernel optimization. In *Proceedings of Machine Learning and Systems (MLSys)*, 2026. arXiv:2511.15915.

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*, 2025.

Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 863–879, 2020.

Yujie Zheng, Zhuo Li, Shengtao Zhang, Jiaqian Wang, Junjie Sheng, Junchi Yan, Weinan Zhang, Ying Wen, Bo Tang, and Muning Wen. Towards cold-start drafting and continual refining: A value-driven memory approach with application to NPU kernel synthesis. In *International Conference on Machine Learning (ICML)*, 2026. arXiv:2603.10846.

A Appendix

A.1 ModelParts-6K construction details

We capture `model.layers.0` from Mistral 7B, Mixtral 8x7B, Qwen3 8B, and DeepSeek V2 Lite, and `model.loop_block` from Ouro 1.4B. These components span sliding, grouped-query, global, and multi-head attention; rotary positional embeddings and query–key normalization; sparse mixture-of-experts routing; weight-tied recurrent depth and sandwich normalization; and low-rank key/value projection. Table 4 reports features of these exact components.

We capture each component for prefill forward, prefill backward, and autoregressive decode forward. Table 5 gives the selected hidden-state dimensions in batch $\times$ sequence $\times$ hidden order. Both prefill configurations contain 4,096 tokens per batch: forward uses a batch of one 4,096-token sequence, while backward uses two 2,048-token sequences. Decode uses four single-token sequences. Hidden width is 4,096 for Mistral, Mixtral, and Qwen3, and 2,048 for Ouro and DeepSeek V2 Lite.

A.2 Operator mix of captured components

Figure 4 compares the ATen operations in the five captured model components across the three capture regimes. To keep the figure legible, it names operations that occur in at least three of the five model layers. Within each semantic category, the *Other* row sums the operations below this threshold; the parenthesized value is the number of distinct operation targets in that aggregate.

A.3 Training task-size distributions

Figure 5 reports the number of operations in both training splits. For ModelParts, we count FX operation nodes in the 5,400 tasks retained after cross-walk deduplication, balanced selection, and the train–evaluation split. NKI-Agent-Ops starts from 6,000 flattened functional references. We count calls in its training references after excluding operations whose ATen schemas identify them as metadata-only aliases. Both counts exclude views, slices, permutes, transposes, graph inputs, attributes, and returns. The ModelParts count also excludes generated Python lifetime markers such as `intermediate = None`, which are not graph operations. Across the 15 balanced ModelParts training groups, the median task contains between four and eight operations, and every task contains at most 17. For Ouro specifically, the prefill-forward, prefill-backward, and decode-forward groups have medians of five, four, and six operations and maxima of 10, 11, and 12, respectively. NKI-Agent-Ops has a median of three operations and a 95th percentile of nine; 1.3% of its training tasks fall in the figure's 18-or-more overflow bin.

A.4 Merkle hashing for cross-model kernel deduplication

ModelParts deduplicates repeated subgraphs using a canonical SHA-256 Merkle hash. In topological order, each node hashes its operation, target, literals, tensor metadata, and the hashes of its predecessors. Source names and module paths are excluded, while inputs and outputs are placed in canonical orders before a version-tagged root hash is computed. Kernels therefore match when they have the same operations, constants, shapes, dtypes, and data dependencies, even when captured from different models. We retain one task per hash and verify the canonically reordered reference against the original captured graph before emission.

A.5 Roofline hardware profiles

AWS reports whole-chip peaks of 181 FP32 TFLOP/s and 2.9 TB/s of HBM bandwidth for Trainium2, which contains eight physical NeuronCores (Amazon Web Services, 2026). The ModelParts partition walk uses these whole-chip values, $r_c = 181$ TFLOP/s and $r_m = 2.9$ TB/s. The reward verifier instead profiles one logical NeuronCore under $\mathrm{LNC} = 2$, comprising two physical NeuronCores, and allocates one quarter of each whole-chip peak to that execution context. It therefore uses $r_c = 45.25$ TFLOP/s and $r_m = 0.725$ TB/s.

Table 4: **Architectural coverage of the five captured components.** Checkmarks identify features present in each component.

Feature	Mistral	Mixtral	Qwen3	Ouro	DeepSeek V2
Sliding/local attention	✓				
Grouped-query attention	✓	✓	✓		
Sparse MoE		✓			
Query–key normalization			✓		
Rotary positional embeddings	✓	✓	✓	✓	✓
Global attention		✓	✓	✓	✓
Weight-tied recurrent depth				✓	
Sandwich normalization				✓	
Multi-head attention				✓	✓
Low-rank KV projection / MLA					✓

Table 5: **Capture dimensions by execution regime.** Entries report batch × sequence × hidden size for each graph.

Regime	Graph	Mistral / Mixtral / Qwen3	Ouro / DeepSeek V2 Lite
Prefill	Forward	$1\times4096\times4096$	$1\times4096\times2048$
Prefill	Backward	$2\times2048\times4096$	$2\times2048\times2048$
Decode	Forward	$4\times1\times4096$	$4\times1\times2048$

A.6 Roofline reward analysis

Figure 6 compares the conditional terminal-reward distributions for the three Qwen3.8-27B PITL settings.

A.7 NKILib optimization trajectories

Figure 7 shows how candidate performance changes within the rollout responsible for each entry in Table 3.

A.8 Correctness verification

The verifier evaluates the executable PyTorch reference F_k on hidden inputs to obtain expected outputs. Candidate outputs must match the reference in count, shape, and dtype; Boolean and integer outputs must also match elementwise. For a floating-point or complex reference output $y = F_k(x)$ of dtype d and candidate output $\hat{y} = f(x)$, each finite element must satisfy

$$|\hat{y}_i - y_i| \le \max\left(\beta_d \ \mathrm{RMS}(y_{\mathrm{finite}}), \epsilon_d\right) + \rho_d |y_i|, \tag{3}$$

where y_{finite} contains the finite reference values. NaNs and signed infinities must occur in the same locations, and every output on every verifier input must pass. Table 6 gives the default constants. These defaults apply to ModelParts and NKI-Agent-Ops.

NKILib tasks preserve the library's task-specific relative and absolute tolerances, r and a. To match NKILib's global-scale comparison, each finite element must satisfy

$$|\hat{y}_i - y_i| \le a + r \max_j |y_j|.$$

For both rules, NaNs and positive or negative infinities must match the reference at the same locations.

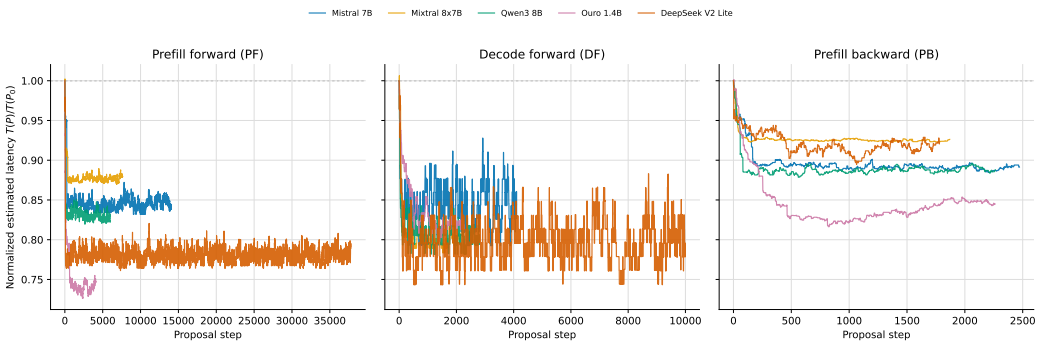


Figure 3: **Normalized ModelParts-6K partition-walk trajectories.** Curves are separated by prefill forward (PF), decode forward (DF), and prefill backward (PB). Each curve shows $T(P)/T(P_0)$ after every post-burn-in proposal, including rejected proposals, for one model component. Colors identify components, and each curve ends when its walk has retained 500 content-distinct kernels.

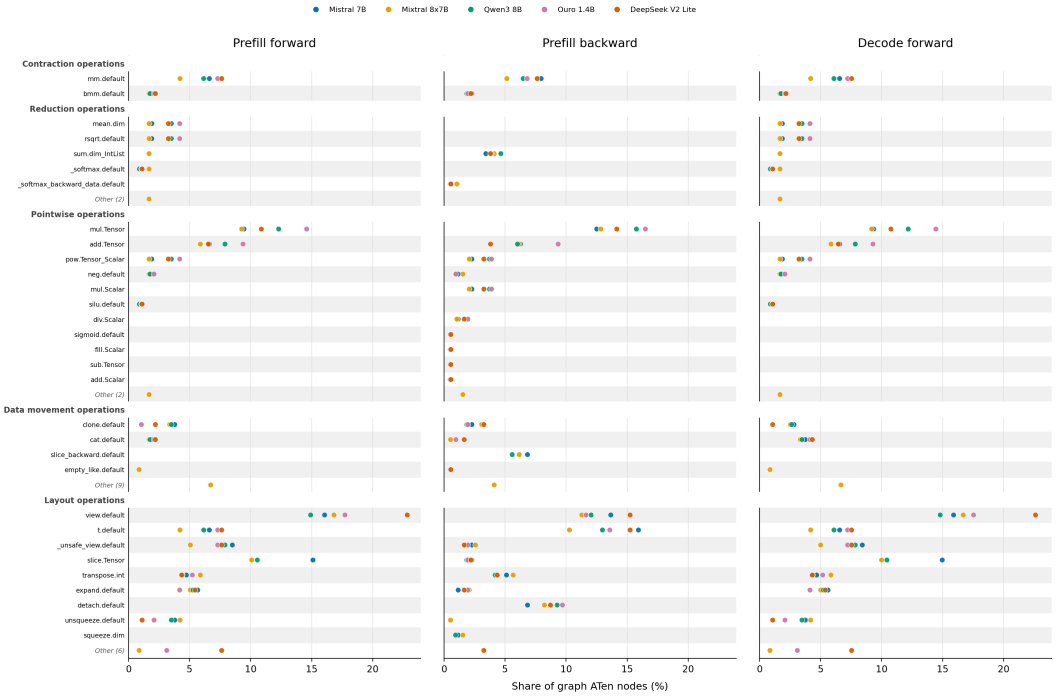


Figure 4: **ATen operation mix of captured model graphs.** Each marker is the percentage of ATen nodes assigned to an operation in one prefill-forward, prefill-backward, or decode-forward graph. Colors identify model components; multiple markers may coincide. Named rows retain operations present in at least three of the five model layers, while each category-specific *Other* row preserves the aggregate mass of the remaining operations.

Table 6: **Default numerical tolerances for generated tasks.** Boolean and integer outputs require exact elementwise equality.

Reference dtype d	ρ_d	β_d	ϵ_d
`bfloat16`	10^{-2}	10^{-3}	10^{-6}
`float32`	10^{-4}	10^{-5}	10^{-6}
`complex64`	10^{-4}	10^{-5}	10^{-6}
`bool`, `uint8`, `int32`, `int64`		exact	

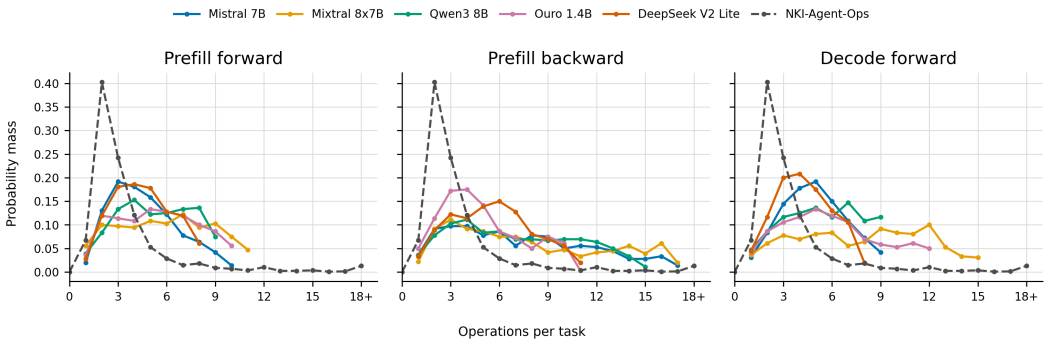


Figure 5: **Training-task size distributions.** Each colored curve is the discrete probability mass over the 360 ModelParts training tasks assigned to one model and capture regime. The dashed gray NKI-Agent-Ops training distribution is repeated in each panel for direct comparison. Before the split, the capacity-constrained ModelParts assignment considers candidate kernels in ascending operation count while preserving exactly 400 tasks from each walk. The final bin aggregates tasks with at least 18 operations.

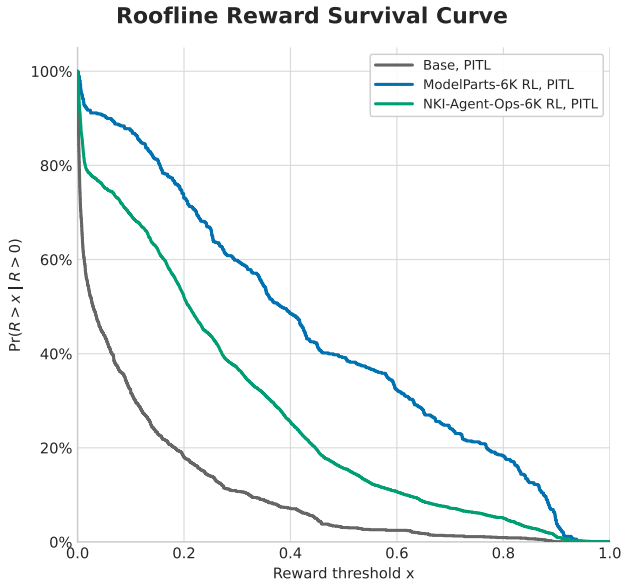


Figure 6: **Roofline reward survival under PITL.** Curves include only terminal `grade-solution` rewards and exclude intermediate `check-solution` calls. ModelParts and NKI-Agent-Ops evaluation rewards are pooled for base Qwen3.8-27B and policies trained on either corpus. Each curve shows $\Pr(R > x \mid R > 0)$.

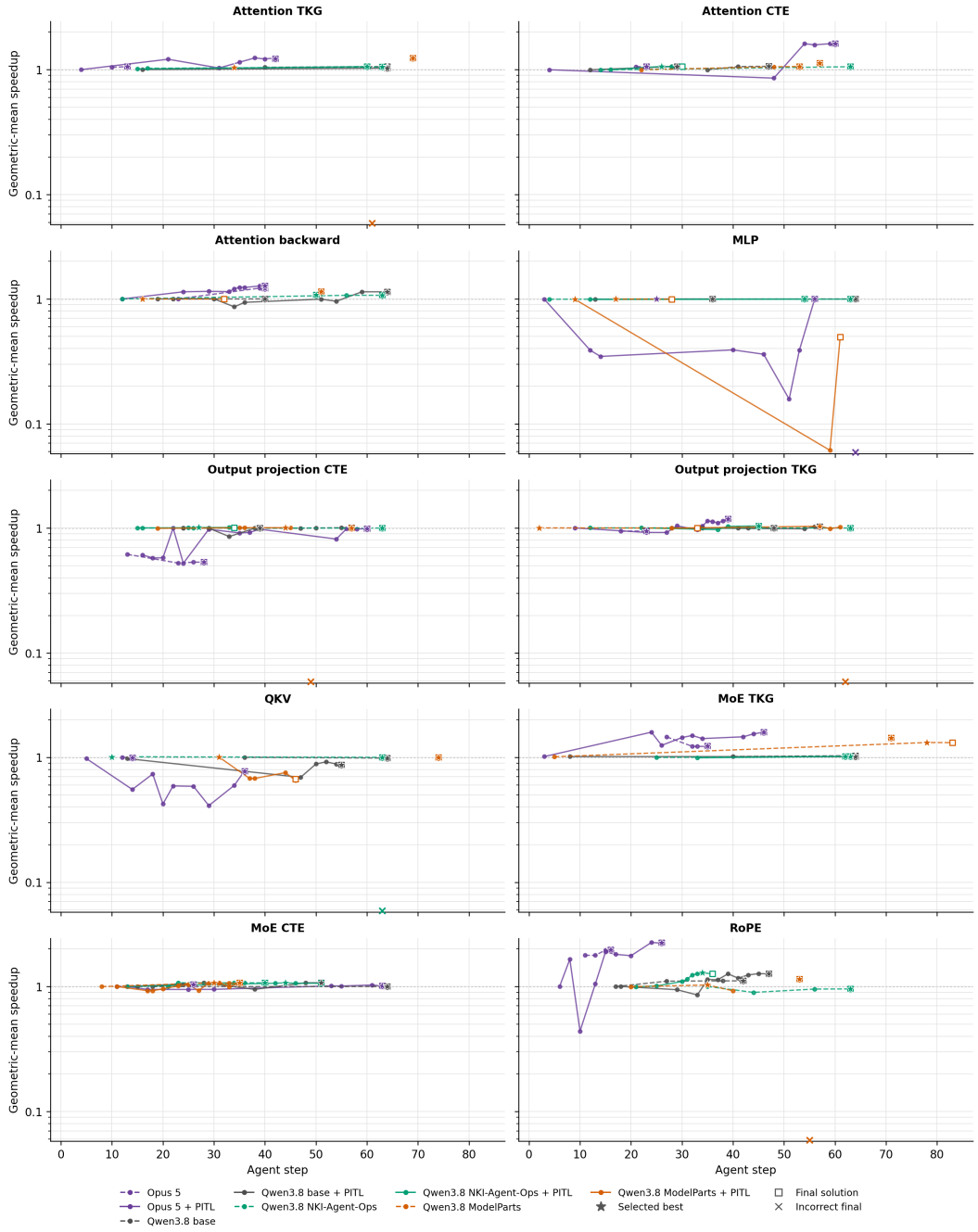


Figure 7: **Best-rollout optimization trajectories on NKILib.** For each task and model configuration, curves show verifier-replayed candidate implementations along the rollout that produced the selected best-of-eight kernel. Stars mark the selected kernel, open squares mark correct final submissions, and crosses mark incorrect final submissions. Speedup is the geometric mean over the three test inputs, relative to the installed expert implementation at $1\times$. Solid curves denote PITL conditions and dashed curves denote non-PITL conditions.

A.9 AGENT PROMPTS

All conditions use the same custom Mini-SWE-Agent system and instance templates. The non-PITL rendering below replaces the task and runtime-specific environment values with placeholders. The subsequent condensed diff summarizes the agent-visible instructions added for profiler-in-the-loop (PITL) access.

```
You are an expert Neuron Kernel Interface (NKI) kernel developer.

Your response must contain exactly ONE bash code block with ONE command (or commands
connected with && or ||).
Format your response as shown in <format_example>.

<format_example>
Your reasoning and analysis here. Explain why you want to perform the action.
```mswea_bash_command
your_command_here
```
</format_example>

Failure to follow these rules will cause your response to be rejected.
```

Listing 1: **System prompt shared by all Mini-SWE-Agent conditions.**

```
<TASK INSTRUCTION>

You can execute bash commands and edit files to implement the necessary changes.

## Recommended Workflow

Follow this workflow step-by-step so that you can quickly iterate on your NKI kernel:

1. Immediately write a complete candidate to `/logs/artifacts/solution.py`.

2. Review the candidate against the reference function and input specification.
3. Make any necessary targeted revisions, then submit your solution.

Submit your solution by issuing the following command:
`echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT`

Do not combine it with any other command. <important>After this command, you cannot continue
 working on this task.</important>

## Important Rules

- Specialize only for the exact input names, shapes, and dtypes provided. Do not support
arbitrary shapes, additional dtypes, generic broadcasting, or unrelated edge cases.

- Do not run full-size CPU simulations or manually compile NIR or NEFF files.
- Avoid background processes, sleeping, and polling.
- Directory and environment variable changes are not persistent. Every action runs in a new
subshell.

<nki_environment>
Operating system: <OS> <RELEASE> <VERSION> <MACHINE>
Neuron SDK: <NEURON SDK VERSION>
NKI version: <NKI VERSION>

Verification target:
- Architecture: Trainium2 (`trn2`)
- Logical NeuronCore configuration: LNC=2
- The task container has no Trainium device.

- The final TRN verifier checks correctness first and profiles latency only when the kernel
is correct.
</nki_environment>
```

Listing 2: **Mini-SWE-Agent instance prompt for non-PITL conditions.** Task- and runtime-specific fields are shown as placeholders.

```
--- no-PITL
+++ PITL
@@ Recommended workflow
-2. Review the candidate against the reference function and input specification.
-3. Make any necessary targeted revisions, then submit your solution.
```

```
+2. Run `check-solution` by itself.
+3. Revise an incorrect kernel from its feedback and run it again. Retry an
+   infrastructure timeout once without changing the kernel.
+4. Submit as soon as the solution is reported correct.
@@ Important rules
+- Before the first call, do not browse installed NKI source, documentation,
+  examples, verifier code, or client code.
@@ Environment
+- `check-solution` compiles the candidate for Trainium2 and sends it to the
+  remote verifier, which profiles latency only after correctness passes.
```

Listing 3: **Instance-prompt changes for PITL access.** The listing shows a condensed diff against the non-PITL prompt.

A.10 REPRESENTATIVE TASK INSTANCES

The following examples reproduce the agent-visible instruction from one task in each evaluation setting. The NKI-Agent-Ops example additionally shows the original class-based CUDA-Agent-Ops program before adaptation, and the NKILib example includes its verifier reference.

A.10.1 MODELPARTS

This example is an RMS-normalization fragment sampled from the prefill-forward graph of the Ouro 1.4B loop block.

````
# Implement a Neuron Kernel Interface (NKI) kernel

Implement an NKI kernel equivalent to this PyTorch function:

```python
import torch


def kernel(primals_1):
    pow_1 = torch.ops.aten.pow.Tensor_Scalar(primals_1, 2)
    mean = torch.ops.aten.mean.dim(pow_1, [-1], True)
    add = torch.ops.aten.add.Tensor(mean, 1e-06)
    rsqrt = torch.ops.aten.rsqrt.default(add)
    mul = torch.ops.aten.mul.Tensor(primals_1, rsqrt)
    return mul
```

Write your solution to: /logs/artifacts/solution.py.

The file must define this exact function signature:

```python
import nki


@nki.jit
def kernel(primals_1):
    # Your NKI implementation.
    ...
```

The verifier generates random inputs matching the following specification, computes expected
 outputs with the PyTorch reference, checks your kernel for correctness, and profiles
latency only if it is correct:

```json
[
  {
    "name": "primals_1",
    "shape": [1, 4096, 2048],
    "dtype": "float32"
  }
]
```
````

Listing 4: **Agent-visible instruction for a ModelParts task.**

A.10.2 NKI-AGENT-OPS

This example composes a thresholding operation with concatenation. The task instruction contains the class-free functional ATen reference produced by adapting the original CUDA-Agent-Ops program shown below it.

````
# Implement a Neuron Kernel Interface (NKI) kernel

Implement an NKI kernel equivalent to this PyTorch function:

```python
import torch

def kernel(x):
    lt = torch.ops.aten.lt.Scalar(x, 0)
````
````

````
    zeros_like = torch.ops.aten.zeros_like.default(x, pin_memory=False)
    where = torch.ops.aten.where.self(lt, zeros_like, x)
    cat = torch.ops.aten.cat.default([where, where], 1)
    return cat
```

Write your solution to: /logs/artifacts/solution.py.

The file must define this exact function signature:

```python
import nki


@nki.jit
def kernel(x):
    # Your NKI implementation.
    ...
```

The verifier generates random inputs matching the following specification, computes expected
 outputs with the PyTorch reference, checks your kernel for correctness, and profiles
latency only if it is correct:

```json
[
  {
    "name": "x",
    "shape": [512, 3, 256, 256],
    "dtype": "float32"
  }
]
```
````

Listing 5: **Agent-visible instruction for an adapted NKI-Agent-Ops task.**

```python
import torch


class Model(torch.nn.Module):
    def __init__(self, num_inputs, num_outputs):
        super(Model, self).__init__()
        self.num_inputs = num_inputs
        self.num_outputs = num_outputs

    def forward(self, x):
        x = torch.where(x < 0, torch.zeros_like(x), x)  # Apply torch.where
        x = torch.cat([x] * self.num_outputs, dim=1)  # Apply torch.cat
        return x


def get_inputs():
    return [torch.randn(512, 3, 256, 256)]  # Increased spatial dimensions to 256x256


def get_init_inputs():
    return [3, 2]  # Number of inputs and outputs for the cat operation
```

Listing 6: **Original class-based CUDA-Agent-Ops program.** This program is transformed into the adapted task above.

A.10.3 NKILIB

This QKV task exposes an expert NKI implementation as the starting point and verifies optimized submissions against a package-faithful PyTorch reference.

````
# Optimize a Neuron Kernel Interface Library (NKILib) Kernel

Inspect the installed NKILib `qkv` kernel and optimize its concrete specialization. The
installed `nkilib` package contains the source implementation and may be imported directly.

A useful way to inspect the implementation is:

```python
import inspect
````

````
from nkilib.core.qkv.qkv import qkv as baseline_kernel

baseline_impl = baseline_kernel.__wrapped__
print(inspect.signature(baseline_impl))
print(inspect.getsource(baseline_impl))
```

This task is a concrete tensor-only specialization of that library kernel. The following
package-faithful adapter defines the exact specialization for this task and has been
validated against the installed baseline across the small, medium, and large representative
inputs:

```python
import nki

from nkilib.core.qkv.qkv import qkv as baseline_kernel


baseline_impl = baseline_kernel.__wrapped__


@nki.jit
def kernel(hidden, weights):
    return baseline_impl(
        hidden,
        weights,
    )
```

Use this adapter as the initial correctness-preserving solution before optimizing it.
Preserve its fixed specialization arguments and output structure. `/workspace/input_spec.
json` describes only the small agent-visible input. Final verification also uses medium and
large inputs with the same tensor interface and dtypes, so derive varying dimensions from
the input tensors instead of hardcoding the visible shapes.

Write the optimized solution to `/logs/artifacts/solution.py`. The file must define this
exact interface:

```python
import nki


@nki.jit
def kernel(hidden, weights):
    # Your optimized NKI implementation.
    ...
```

The solution must produce the same outputs as the installed NKILib baseline kernel across
representative inputs. The verifier compares it with a PyTorch reference validated against
that baseline. Only correct solutions are profiled; lower latency is better.
````

Listing 7: **Agent-visible instruction for an NKILib QKV optimization task.**

```python
import torch

from nkilib.core.qkv.qkv_torch import qkv_torch_ref


def generate_inputs(input_spec, seed):
    generator = torch.Generator().manual_seed(seed)
    return {
        item["name"]: torch.randn(
            item["shape"],
            dtype=torch.float32,
            generator=generator,
        ).to(getattr(torch, item["dtype"]))
        for item in input_spec
    }


def kernel(hidden, weights):
    return qkv_torch_ref(
        input=hidden,
        fused_qkv_weights=weights,
    )["out"].to(hidden.dtype)
```
```

Listing 8: **PyTorch correctness reference for the NKILib QKV task.** The reference is validated against the expert implementation.