HiPO: A High-Throughput Playground and a Clean Policy-Gradient Baseline for HIP Kernel Optimisation

Anonymous ACL submission

Abstract

Public RL baselines for AMD HIP kernel optimisation lag well behind those for CUDA, Triton, and PyTorch op-level kernel stacks. It remains open whether an executable-reward agentic RL recipe transfers to HIP. We address this gap with **HiPO**, an end-to-end design with two components. **The HiPO playground** is a high-throughput, asynchronous evaluator that reuses one compile/run/match core for online RL reward, in-rollout feedback, and offline benchmark scoring; it lifts batch-100 evaluator wall-time by $\mathbf{2.32}\times$ and cuts warm-reference-cache time by $\mathbf{43}\%$. **The HiPO RL recipe** is a clean policy-gradient and agentic-thinking baseline trained on that playground. We evaluate HiPO-8B on KernelBench-HIP against a GPT-5 best-of-5 HIP reference at k=246. At n=1 HiPO clears parity on **99** kernels vs. 38 for the base (F@1.0: $\mathbf{40.2}\%$ vs. 15.4%, $\Delta = +\mathbf{61}$ kernels). The F@1.05 lead opens with rollout: **41** vs. 36 at n=8 (+5 kernels) and **47** vs. 45 at n=16 (+2 kernels), with paired W/D/L $\mathbf{19/218/9}$ at $\pm 5\%$. The canonical headline is a $\mathbf{1.461}\times$ Top-10-by-HiPO geomean at n=16, against $1.257\times$ for the base ($\Delta = +\mathbf{0.204}$, $10/10$ kernels at F@1.05). The peak audited per-kernel speedup reaches $\mathbf{1.35}\times$ on a shared-memory–tiled depthwise convolution (L1). To our knowledge, this is the first published RL-trained model to beat a frontier-LLM reference on a public HIP kernel benchmark.

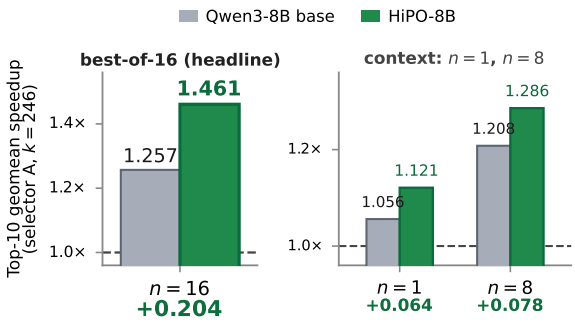


Figure 1: **HiPO-8B on the Top-**10 **selector-A cut of KernelBench-HIP** (k=246, parity dashed = GPT-5 best-of-5; selector B co-reported in Tab. 4). **Left:** best-of-16 geomean $\mathbf{1.461}\times$ vs. $\mathbf{1.257}\times$ ($\Delta = +\mathbf{0.204}$; $\mathbf{10/10}$ kernels at F@1.05). **Right:** Top-10 geomean at the two shown context rollouts, where HiPO leads at $n \in \{1, 8\}$. Selector-independent headline (F@1.0 at n=1, all 246 kernels): $38 \rightarrow \mathbf{99}$ kernels above parity ($\Delta = +\mathbf{61}$; Tab. 1).

1 Introduction

GPU kernels account for the bulk of runtime and energy cost in large-model training and inference, so even modest per-kernel speed-ups translate into substantial savings at production scale. *Executable rewards* now make this RL loop tractable: judging code by what happens when it compiles, runs, and matches the reference. Yet the published RL post-training and agentic kernel-LLM literature stays almost entirely on CUDA, Triton, and PyTorch op-level kernel stacks (Li et al., 2026; Liu et al., 2026; Baronio et al., 2025; Li et al., 2025c; Lange et al., 2025; Zhang et al., 2025). AMD HIP, a major open GPU stack for production AI workloads, has received comparatively little public RL post-training attention. Whether an executable-reward agentic recipe transfers cleanly to HIP therefore remains open.

Existing work on kernel optimisation partitions into four directions, anchored by KernelBench (Ouyang et al., 2025). These directions are search-based generation (Cao et al., 2026; Guo et al., 2025; Zhang et al., 2025), multi-agent write/critique loops (Wei et al., 2025; Lei et al., 2025; Wang et al., 2025a), domain-specific kernel models (Fisches et al., 2025; ByteDance Seed, 2025; Paliskara and Saroufim, 2025), and CUDA/Triton RL post-training (Li et al., 2026; Liu et al., 2026; Baronio et al., 2025; Li et al., 2025c). None of these directions targets HIP at the full-translation-unit RL-training level. The closest non-CUDA reference points are Geak (Wang

et al., 2025a,b), an agentic Triton/HIP kernel framework on AMD GPUs (not RL-trained), and emerging-hardware benches such as KernelCraft and NPUEval (Nie et al., 2026; Kalade and Schelle, 2025). The full taxonomy and AMD/non-CUDA gap are treated in §6.

We formalise the gap as a setup that poses two coupled questions. A task $x{=}(\text{spec}, c_0, h, d)$ comprises a PyTorch reference functional, a HIP reference file c_0 from a frontier-LLM best-of-5, target hardware h, and a test distribution d. A policy π emits a complete HIP translation unit c. The evaluator returns hard executable validity $V(c){=}\mathbf{1}[\text{parse} \wedge \text{compile} \wedge \text{run} \wedge \text{match}]$ and, conditional on V, a measured speedup $s(c){=}T(c_0; h, d)/T(c; h, d)$. Training maximises $\mathbb{E}_{x,c\sim\pi(\cdot|x)}[V(c) \cdot g(s(c))]$ for a monotone reward shaper g. **Q1 (evaluator):** can it deliver executable feedback at RL scale on HIP? **Q2 (policy):** can a policy-gradient recipe converge under that sparse, easily-hacked signal and beat a frontier-LLM HIP reference?

A positive answer is consequential: it places the AMD HIP systems layer underlying production AI within reach of executable-reward RL post-training. The systems lever is the playground: one scalable evaluator serves online reward, in-rollout tool calls, and offline scoring, so training and benchmarking share the same executable semantics. After decoupling a compile stage that costs $\mathbf{3.3}\times$ the run stage, the same evaluator delivers a $\mathbf{2.32}\times$ batch-100 wall-time lift and a $\mathbf{43}\%$ warm-cache reduction. Concretely, on KernelBench-HIP ($k{=}246$), HiPO-8B clears GPT-5 parity on **99** kernels at $n{=}1$ vs. 38 for the Qwen3-8B base ($\Delta{=}{+}\mathbf{61}$ kernels). HiPO-8B then leads F@1.05 at every rollout from $n{=}8$ upward (**41** vs. 36 at $n{=}8$; **47** vs. 45 at $n{=}16$). On the Top-10 selector A cut at $n{=}16$, HiPO-8B reaches $\mathbf{1.461}\times$ geomean speedup vs. $1.257\times$ for the base ($\Delta{=}{+}\mathbf{0.204}$, $\mathbf{10/10}$ kernels at F@1.05; Fig. 1). We are not aware of a prior RL policy achieving this margin against a frontier-LLM HIP reference on a public benchmark.

Our contributions answer Q1 and Q2 jointly:

- **The HiPO playground** (§2): a high-throughput, asynchronous, scalable evaluator for HIP kernels that supplies executable rewards at RL scale and is reused online (training reward), in-rollout (agentic tool calls), and offline (KernelBench-HIP).
- **The HiPO RL recipe** (§3): a clean, robust policy-gradient and agentic-thinking baseline for full-file HIP generation. The body keeps one HIP-specific design choice, an action contract whose parse→compile→run→match→copy gates each emit first-class gradient signal (§3.1), on top of a standard GRPO recipe with bounded detached entropy bonus and asymmetric upside-biased clip (§3.2). Trained on the playground and locked as HiPO-8B.

The remainder of the paper is organised as follows. §2 details the playground; §3 details the RL recipe; §4–§5 report KernelBench-HIP results; §6 positions HiPO against prior work.

2 The HiPO Playground

The HiPO playground is one evaluator service shared across online RL reward, in-rollout tool calls, and offline KernelBench-HIP scoring (Fig. 2), so the compile/run/match semantics seen by the policy during training are identical to those used at benchmark time. Three async design pillars make its throughput high enough to drive online executable-reward RL on full HIP translation units.

Parallel compilation amortises the dominant cost. Compile dominates the per-candidate budget on both the candidate and reference paths, so HiPO pins compilation to a CPU pool of **16** *outer* compile workers $\times$ **4** *inner* jobs each: the outer dimension parallelises across kernels in the batch, the inner fan-out hides per-translation-unit `hipcc` cold start by overlapping front-end / back-end passes on the same kernel. The combined $\sim$64 effective compile slots saturate the CPU pool independently of GPU availability, so the cost asymmetry between compile and run is removed before any GPU work begins.

Async CPU–GPU dispatch keeps both pools busy. A master process round-robins evaluation tasks across local and remote GPU slots; each slot is guarded by an `asyncio.Semaphore(1)` for strict one-kernel-per-GPU exclusivity, which avoids cross-tenant perf contamination of the reward signal. The dispatcher is `asyncio`-coroutine–based, so the run stage of batch t and the compile stage of batch $t{+}1$ proceed concurrently on their respective pools rather than sequencing through a single chain. In steady state neither pool starves the other; head/tail handoff windows are amortised

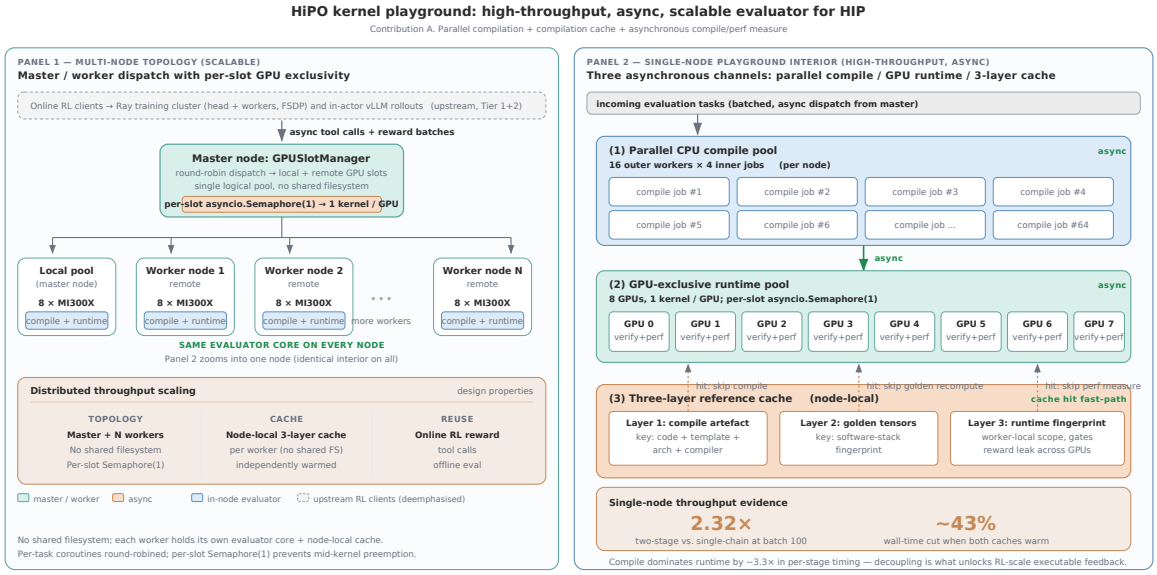


Figure 2: **HiPO kernel playground.** High-throughput async evaluator built on three pillars: a *parallel CPU compile pool* (16×4 jobs) decoupled from a *GPU-exclusive runtime pool* via async dispatch; a *reference cache* with two layers on by default (compile artefact + golden tensors) and an optional worker-local perf-fingerprint layer (off in production); and a *master/worker dispatcher* round-robining tasks across local and remote GPU slots. The same core serves online RL reward, in-rollout tool calls, and offline KernelBench-HIP, lifting batch wall-time $\mathbf{2.32}\times$ at batch 100 and cutting reference-cache time by $\mathbf{43}\%$ via warm caching (§5.5).

over the batch, which is the structural reason per-candidate wall time *decreases* with batch size rather than plateauing (quantified in §5.5).

Three-layer reference cache, two enabled by default. Each reference path queries three caches with disjoint keys: (i) *compile-artefact* keyed on source hash (returns the linkable object on hit; skips the compile step), (ii) *golden tensors* keyed on (problem-id, input shape) (returns blessed reference outputs; skips the reference run+match), and (iii) *worker-local perf fingerprint* keyed on (problem-id, GPU UUID) (returns reference wall-time without re-running). Layers (i)–(ii) ship enabled by default in both training and eval; (iii) is intentionally disabled in production so cross-GPU perf cannot leak into the reward signal, and is enabled only for the throughput benchmark of Fig. 7c. End-to-end wall-time evidence for all three pillars is in §5.5; implementation entry points are in App. D.

3 Policy-Gradient Baseline: HiPO-8B

The baseline is deliberately standard. The only HIP-specific component we keep in the body is the action contract (§3.1) that turns the gated reward into a first-class executable signal; the GRPO recipe with entropy-shaped advantage and asymmetric clip is in §3.2, with full derivation in App. H. The training loop is Alg. 1; hyperparameters are in App. E.

3.1 Action contract under verifiable feedback

The model emits a JSON object {`thought`, `code`} with `code` a complete HIP translation unit. Reward is a sequential cascade (compile, run, match, copy-detect, perf), short-circuited to 0 on the first failure:

$$r(c) = \begin{cases} 0, & \neg V, \\ r_{\text{copy}}, & V \wedge C, \\ r_{\text{ok}} + \min(s, s_{\max}) + b, & V \wedge \neg C. \end{cases} \quad (1)$$

Validity V short-circuits compile $\wedge$ run $\wedge$ match. The copy gate $C = \mathbf{1}[d_{\text{DTW}}(c, c_0) < d_{\text{copy}}(L_c)]$ uses a token-DTW distance with adaptive threshold in $[0.05, 0.15]$ (App. B). The perf branch uses measured speedup s, clip $s_{\max} = 10$, and a small above-reference bonus b. Defaults: $r_{\text{ok}} = 0.3$, $r_{\text{copy}} = 0$. Sequencing keeps the copy detector from contaminating the gradient with rewards on broken kernels (Fig. 3).

3.2 GRPO with entropy-shaped advantage and asymmetric clip

The reward (Eq. 1) is sparse, since most candidates fail match, and easily hacked via copies of the reference. We address both with a standard GRPO

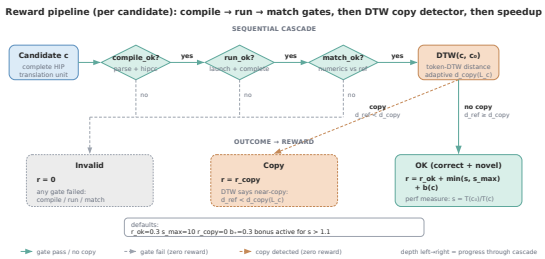


Figure 3: **Reward pipeline (per candidate).** Cascade of three gates (*compile* → *run* → *match*), each routing failures to $r{=}0$, followed by a token-DTW copy detector against the reference c_0. Outcomes partition candidates into Invalid / Copy / OK; only OK reaches perf measurement and the clipped-speedup-with-bonus reward of Eq. 1.

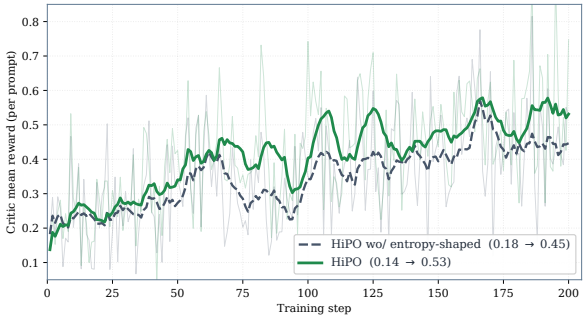


Figure 4: **HiPO-8B training reward curve.** Solid green: HiPO-8B per-prompt mean reward over 200 steps; dashed slate: sibling run differing only in rollout width and PPO entropy coefficient (reference only). Bold = 10-step trailing mean. HiPO-8B climbs monotonically out of cold-start ($0.14 \rightarrow \mathbf{0.53}$ smoothed, $0.14 \rightarrow 0.75$ raw at step 200), with no collapse and no saturation; the locked checkpoint of Tab. 1 is taken from this run.

recipe: a group-normalised advantage (Eq. 5) and length-invariant aggregation (Eq. 6–7), both written out in App. H. Two design choices stay in the body: a bounded detached entropy bonus on the advantage, and an asymmetric clipped surrogate biased toward upside exploration. The entropy-shaped advantage adds a detached, capped entropy bonus to the group-normalised advantage $\hat{A}^{(i)}$:

$$\tilde{A}_t^{(i)} \;=\; \hat{A}^{(i)} \;+\; \min\Big(\alpha \cdot \mathrm{sg}\big[H_\theta(s_t^{(i)})\big],\; |\hat{A}^{(i)}|/\kappa\Big), \tag{2}$$

with $\alpha{=}0.4$, $\kappa{=}2$, and $\mathrm{sg}[\cdot]$ a stop-gradient, so the bonus contributes only to the magnitude of an already-signed advantage and never flips its sign. With token-level importance ratio $u_t^{(i)} \equiv \pi_\theta(y_t^{(i)}|s_t^{(i)})/\pi_{\theta_{\text{old}}}(y_t^{(i)}|s_t^{(i)})$, the token surrogate is

$$\mathcal{L}_t^{(i)} \;=\; -\min\Big(u_t^{(i)}\,\tilde{A}_t^{(i)},\; \mathrm{clip}\big(u_t^{(i)}, c_{\text{lo}}, c_{\text{hi}}\big)\,\tilde{A}_t^{(i)}\Big), \tag{3}$$

with $(c_{\text{lo}}, c_{\text{hi}}){=}(0.8, 1.4)$ (Shao et al., 2024; Yu et al., 2025). The asymmetric clip ($\epsilon_{\text{lo}}{=}0.2$, $\epsilon_{\text{hi}}{=}0.4$) is exploration on the upside: larger positive than negative updates, so the policy commits to high-advantage rewrites without symmetric regression on rare high-magnitude losses.

4 Experimental Setup

Models compared. **Baseline** is Qwen3-8B (no RL); **HiPO-8B** is the entropy-shaped GRPO policy of §3 at the locked checkpoint. Both runs share one orchestrator and differ only in the model checkpoint; every React, decoding, and rollout flag is identical.

Datasets and reference baselines: KernelBench-HIP (eval) and HiPO-9k (training). Both corpora are built by a common two-stage PyTorch-to-HIP pipeline: each task is first standardised into a function-style PyTorch reference, then a HIP candidate is synthesised by best-of-5 GPT-5 sampling on performance and admitted only if it parses, compiles, runs, and numerically matches the PyTorch reference under random inputs (`atol/rtol`$=10^{-4}/10^{-3}$), with the fastest survivor retained. **KernelBench-HIP** ports the Stanford KernelBench (Ouyang et al., 2025) PyTorch-to-CUDA suite of 250 tasks (L1: 100 atomic operators, L2: 100 fused operators, L3: 50 full architectures); 247 pass, and excluding `23_Softmax_base` (verifier hole) leaves $k{=}246$ (98/100/48 across L1/L2/L3 after dedup), with an origin-vs-origin sweep giving noise floor $s \in [0.97, 1.04]$ so any HiPO-8B candidate at $s > 1.0\times$ beats the best of five GPT-5 attempts. **HiPO-9k** applies the same pipeline to GPUMODE/KernelBook (Paliskara and Saroufim, 2025) for RL post-training.

Evaluation configuration. KernelBench-HIP (L1/L2/L3, all 246 kernels per rollout sweep). The model emits a complete HIP file under the strict JSON `thought/code` contract. At inference time only we use three sequential turns with best-of-n per turn; later turns receive the previous candidate, correctness/performance feedback, and failure evidence. We sweep $n \in \{1, 4, 8, 16\}$ as rollouts; candidates are selected by best-history.

Training configuration. Headline settings: Qwen3-8B base, group-normalised GRPO with $G{=}8$ rollouts, asymmetric clip

$(c_{\text{lo}}, c_{\text{hi}})=(0.8, 1.4)$, entropy bonus $\beta_{\text{ent}}=0.02$, length-invariant $T_{\max}=16$K token-sum aggregation, correctness-gated copy-suppressed reward ($r_{\text{ok}}=0.3$, $s_{\max}=10$, $r_{\text{copy}}=0$), single-turn React rollouts (one JSON `thought`/`code` message per rollout). The multi-turn React loop above is an inference-time scaling protocol only; training optimises single-turn responses against the rollout-level reward. The full hyperparameter table (optimisation, decoding, loss, reward, React, data filtering) is in App. E.

Metrics. Per-kernel speedup is DTW-dedup before aggregation. The primary head-to-head is Tab. 1: per (level, n) cell we report paired solved counts, paired geomean (missing $\rightarrow 1.0\times$), Fast@p counts at $p \in \{1.0, 1.05, 1.1, 1.2, 1.5\}$ on the $n=16$ column, and a paired W/D/L on the $n=16$ column. **Fast@p** (*F@p*) counts the kernels whose DTW-dedup measured speedup is $\geq p$, rendered as count (pct%) over the row's k denominator (e.g. 47 (19.1%)). **W/D/L** pairs HiPO against the baseline per kernel at $\pm 5\%$: HiPO wins (W) if its speedup beats the baseline's by $> 5\%$, draws (D) if within $\pm 5\%$, loses (L) otherwise. Tab. 2 and Tab. 4 re-aggregate the same data on top-k subsets ($k \in \{10, 20, 30\}$) under selector A (top-k by HiPO, performance-critical) and selector B (top-k by max, headroom-controlled); definitions in App. I.

5 Results

5.1 Headline result

HiPO-8B beats the GPT-5 best-of-5 HIP reference on KernelBench-HIP at every rollout budget we tested, and the lead is consistent across the selector-free count, the per-kernel paired view, and the performance-critical top-k cut. At $n=1$, HiPO clears parity on **99** of 246 kernels against 38 for the Qwen3-8B base ($\Delta = +\mathbf{61}$ kernels; $15.4\% \rightarrow \mathbf{40.2}\%$). At $n=16$, HiPO leads F@1.05 by **2** kernels (**47** vs. 45) and pairs **19/218/9** wins/draws/losses at $\pm 5\%$. On the performance-critical Top-10 selector-A cut at $n=16$, HiPO posts a $\mathbf{1.461}\times$ paired geomean against $1.257\times$ for the base ($\Delta = +\mathbf{0.204}$, **10/10** kernels at F@1.05; Tab. 1, Fig. 1). To our knowledge this is the first published RL policy to beat a frontier-LLM reference on a public HIP kernel benchmark.

5.2 Main head-to-head

Tab. 1 carries two row blocks (Qwen3-8B base, HiPO-8B) with identical column structure, so every HiPO cell sits directly below its baseline counterpart; the complementary top-k cuts are in Tab. 2 and Tab. 4.

Per-sample skill is where the policy moves the needle. The clearest signal is the selector-free $n=1$ count. HiPO produces **99** above-reference kernels on the first sample against 38 for the base ($\Delta = +\mathbf{61}$ kernels, $2.6\times$ as many); paired geomean shifts $0.927 \rightarrow \mathbf{0.969}$ on the same draw. The shift is a broad jump in valid above-parity candidates, not a single geomean spike.

Modest rollout already opens the F@1.05 lead. At $n=4$ HiPO trails F@1.05 by one kernel; at $n=8$ the gap flips to $+\mathbf{5}$ kernels (**41** vs. 36, $\mathbf{16.7}\%$ vs. 14.6%) and F@1.1 lifts to **17** vs. 11 kernels. A single doubling of the rollout budget is enough to convert the per-sample skill of the previous paragraph into a parity-clearing lead.

At $n=16$ HiPO's advantage is concentrated but clear. HiPO leads the stricter F@1.05 count by **2** kernels (**47** vs. 45) and pairs **19/218/9** at $\pm 5\%$ on ALL-246. The sharper signal is the performance-critical Selector-A cut: on Top-10, HiPO reaches $\mathbf{1.461}\times$ geomean vs. $1.257\times$ for the base ($\Delta = +\mathbf{0.204}$), with all ten kernels above F@1.05. The broad aggregate geomean remains tail-sensitive because a few baseline mega-wins dominate the all-kernel average.

HiPO's advantage concentrates on performance-critical kernels. Selector A (Tab. 2) shows a positive Top-10 HiPO lead at every rollout budget in the double-check audit, rising to the headline Top-10@$n=16$ ($\mathbf{1.461}\times$ vs. $1.257\times$, $\Delta = +\mathbf{0.204}$). Selector B (Tab. 4) is the anti-cherry-pick check that caps the upper bound on cherry-picking rather than measuring HiPO skill; we report both.

5.3 Parallel and serial scaling

Parallel and serial scaling are complementary. Under the shared $n \times 3$-turn protocol (Fig. 5), baseline parallel scaling buys coverage but not per-sample quality: solve count climbs $75 \rightarrow 91 \rightarrow 98$ across $n=1 \rightarrow 8 \rightarrow 16$ while dedup geomean barely moves ($0.93 \rightarrow 1.01 \rightarrow 1.03$); HiPO-8B lifts both axes (headline numbers in §5.1). Serial refinement is the complementary axis: the turn-1 $\rightarrow$ turn-3 lift shrinks monotonically with rollout size for HiPO-8B ($\Delta_{\text{serial}} = +\mathbf{19.8}$ pp at $n=1$, $+14.6$ pp at $n=4$, and only $+\mathbf{7.3}$ pp at $n=16$; Fig. 6). Extra turns matter most where the first turn is furthest from

Level	k	n=1 Solved	n=1 Gmean	n=4 Solved	n=4 Gmean	n=8 Solved	n=8 Gmean	n=16 (**best-of-**16) Solved	n=16 Gmean	F@1.0	F@1.05	F@1.1	F@1.2	F@1.5	W/D/L @ n=16
GPT-5 ref. (best-of-5)	—	—	1.000	—	1.000	—	1.000	—	1.000	—	—	—	—	—	—
Block A: Qwen3-8B base (3-turn React at inference, DTW-dedup, GPT-5 ref. denominator; F@p as count (pct%))															
L1	98	31	0.888	58	0.991	86	1.031	91	1.034	75 (76.5%)	22 (22.4%)	8 (8.2%)	5 (5.1%)	0 (0.0%)	—
L2	100	48	0.991	82	1.030	95	1.038	97	1.042	82 (82.0%)	18 (18.0%)	4 (4.0%)	2 (2.0%)	1 (1.0%)	—
L3	48	23	0.883	39	0.989	47	1.010	48	0.997	26 (54.2%)	5 (10.4%)	1 (2.1%)	0	0	—
ALL-246	246	102	0.927	179	1.006	227	1.030	236	1.030	183 (74.4%)	45 (18.3%)	13 (5.3%)	7 (2.8%)	1 (0.4%)	—
Block B: HiPO-8B (same scaffolding; cells where HiPO leads/trails baseline are HiPOGreen/BaselineGray)															
L1	98	**45**	**0.919**	**77**	0.984	**91**	**1.038**	**93**	1.013	**87** (88.8%)	**25** (25.5%)	**12** (12.2%)	**6** (6.1%)	0	**12/80/6**
L2	100	**59**	**1.008**	**85**	1.018	95	1.029	95	1.032	**92** (92.0%)	14 (14.0%)	4 (4.0%)	1 (1.0%)	1 (1.0%)	3/94/3
L3	48	**35**	**0.997**	**42**	**1.000**	45	**1.013**	46	**1.010**	**35** (72.9%)	**8** (16.7%)	**2** (4.2%)	0	0	**4/44/0**
ALL-246	246	**139**	**0.969**	**204**	1.001	**231**	1.029	234	1.020	**214** (87.0%)	**47** (19.1%)	**18** (7.3%)	7 (2.8%)	1 (0.4%)	**19/218/9**

Table 1: **Primary head-to-head on KernelBench-HIP: Qwen3-8B base vs. HiPO-8B, per-level pair-block view.** Speedups are DTW-dedup against the GPT-5 best-of-5 origin reference; `23_Softmax_base` (verifier hole) is excluded throughout the aggregates, leaving $k=98$ for L1 and 246 overall; a separate fused-operator verifier-hole case is excluded from single-kernel headline claims. Gmean is the paired geomean with missing-cell rule missing$\rightarrow 1.0\times$ so baseline and HiPO geomean denominators are identical on every (level, n) cell. *Solved* counts the V=1 candidates (out of k). F@p cells render as count (pct%) at the row's k denominator (e.g. ALL-246, n=16, HiPO F@1.05 is **47** kernels at $\geq 1.05\times$, 19.1% of 246). *W/D/L* compares HiPO vs. baseline at $\pm 5\%$ on the n=16 column.

Setup	n	n_{pool}	**Top-**30 **(A) by HiPO** B-geo	H-geo	Δ	**Top-**20 **(A) by HiPO** B-geo	H-geo	Δ	**Top-**10 **(A) by HiPO** B-geo	H-geo	Δ
Single-shot	1	96	1.019	**1.047**	+0.028	1.033	**1.069**	+0.035	1.056	**1.121**	+0.064
Best-of-4	4	80	1.040	**1.093**	+0.052	1.061	**1.135**	+0.074	1.103	**1.225**	+0.122
Best-of-8	8	93	1.136	1.129	−0.007	1.182	**1.183**	+0.000	1.208	**1.286**	+0.078
Best-of-16	16	74	1.163	**1.195**	+0.032	1.212	**1.277**	+0.066	1.257	**1.461**	+0.204

Table 2: **Top-k deep cut (selector A: top-k by HiPO).** Secondary view from the perf-100 double-check audit. Selector A ranks each kept pool by HiPO's best-of-checkpoints speedup, where $H = \max(H^{\text{fill}}_{140}, H^{\text{fill}}_{160})$ after the missing$\rightarrow 1.0\times$ fill. Columns *B-geo* (**B**aseline geomean, Qwen3-8B base) and *H-geo* (**H**iPO-8B geomean) are paired geomean speedups on the indicated kernel subset (see App. I); Δ = H-geo − B-geo is HiPO's geomean lead over the base. The canonical headline is top-10 at n=16: $\mathbf{1.461}\times$ vs. $\mathbf{1.257}\times$ ($\Delta = +\mathbf{0.204}$).

solved, and matter little once n=16 already covers $\sim 90\%$ of kernels; the inference-time $n \times 3$ React protocol exploits both axes.

5.4 Per-level reading

Each KernelBench level tells a sharper version of the same story: HiPO improves where there is structural headroom and matches a saturated baseline elsewhere.

L1 (atomic operators): HiPO wins outright on solve and F@1.05. At n=1 paired geomean is $\mathbf{0.919}\times$ vs. $0.888\times$ with **45** vs. 31 solves; at n=16 F@1.05 is **25** vs. 22 kernels out of 98 ($\mathbf{25.5}\%$ vs. 22.4%) with W/D/L $\mathbf{12/80/6}$; peak walk-through win is $\mathbf{1.35}\times$ on a tiled depthwise conv (App. G).

L2 (fused operators): saturated baseline; HiPO matches geomean under limited clean headroom. HiPO ties the base on geomean ($1.032\times$ vs. $1.042\times$ at n=16) and trails F@1.05 count by **4** kernels (14 vs. 18). The audited single-kernel headline therefore comes from L1, where the depthwise convolution reaches $\mathbf{1.35}\times$ under the stricter verifier.

L3 (full architectures): parity cleared; full-network absolute speedups remain the open frontier. HiPO reaches $\mathbf{1.010}\times$ geomean at n=16 on 46 kernels against baseline $0.997\times$; F@1.05 rises from 5 to **8** kernels (+3; 16.7% vs. 10.4%). The L3 absolute-speedup frontier is the explicit next iteration of the playground; we frame it as an open community direction in §7.

5.5 HiPO throughput evidence

The HiPO playground delivers **2**× to **3**× throughput over a single-chain evaluator at every batch size we tested, and the gain *widens* with batch — exactly the regime that matters for online RL where one

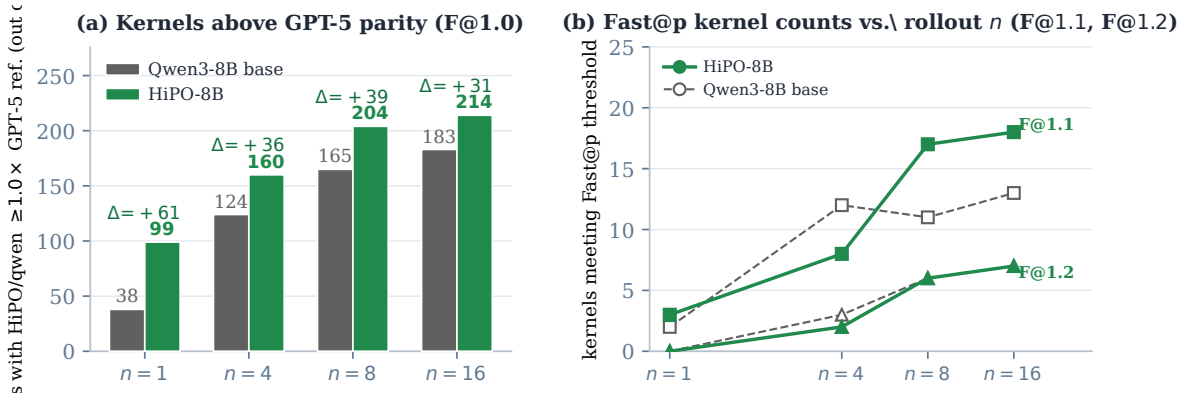


Figure 5: **Parallel scaling: HiPO-8B vs. GPT-5-strong baseline on KernelBench-HIP** ($k = 246$**, Softmax excluded).** *Left:* best-history F@1.0 kernel count vs. rollout n, with HiPO−baseline gap above each pair ($\Delta =$ **+61**/+ 36/+ 39/+ 31 at $n = 1/4/8/16$). *Right:* Fast@p counts at $p \in \{1.1, 1.2\}$ vs. n; HiPO-8B (solid green) dominates Qwen3-8B base (dashed grey) at every (n, p) cell.

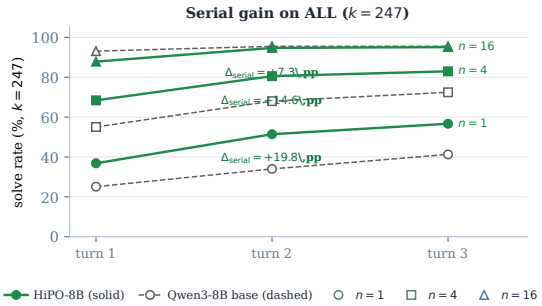


Figure 6: **Serial scaling: per-turn solve rate at fixed rollout n on ALL kernels ($k = 247$).** Solid green: HiPO-8B; dashed grey: Qwen3-8B base. Marker shape: ○ n=1, □ n=4, △ n=16. Turn-1 → 3 slope is steepest at n=1 and flattens monotonically as n grows: extra turns buy most where rollout coverage is lowest.

training step issues hundreds of candidates. Fig. 7 supplies three pieces of evidence: the compile-vs-run timing budget that justifies the two-stage decoupling (a); the batch-positive throughput curve (b); and the layered reference cache (c).

Compile dominance justifies the two-stage decoupling. Per-stage timing at batch 50 shows compile is $\sim$**3.3**$\times$ the run cost on both candidate and reference paths (29.2 s vs. 8.8 s; Fig. 7a) — the dominant-cost asymmetry that motivates the parallel-compile pool and async run/compile overlap of §2. This is the systems bottleneck that a naive kernel-RL loop usually hides: a GPU runtime slot cannot produce reward until CPU compilation has finished, so a serial evaluator turns scarce GPU time into waiting time. HiPO therefore treats compilation as a saturable CPU service and runtime as an exclusive measurement service. The design preserves one-kernel-per-GPU timing integrity while keeping the GPU fed with already-compiled candidates, which is precisely the separation needed for executable reward rather than offline-only benchmarking.

Throughput scales positively with batch – the system gets faster as load grows. Wall-time speedup over the single-chain baseline rises from **2.25**$\times$ at batch 25 to **2.32**$\times$ at batch 100 (Fig. 7b), counter to naive pipelines where larger batches expose tail stragglers. The mechanism (handoff windows amortised across candidates) is the async-dispatch story of §2; because the same playground binary serves online-RL reward, in-rollout tool calls (`HIP_TOOL_*`), and offline KernelBench-HIP evaluation, the per-candidate cost reduction transfers 1:1 to training and inference-eval wall-time and dominates the non-LLM cost in our locked HiPO-8B run. The important point is not only the absolute speedup, but its direction with load: GRPO steps and best-of-n inference both ask the evaluator for many independent candidates on the same hardware. A batch-positive curve means additional rollouts improve search coverage without linearly expanding systems overhead, which makes the playground a reusable RL substrate rather than a one-off evaluation script.

Reference cache adds another −43% when warm. On top of the pipeline gain, the three-layer reference cache (§2) cuts batch-8 wall time from 44.7 s (cold) to 25.5 s once warm (Fig. 7c). The cached objects are reference-side invariants: compile artefacts and golden tensors depend on the task specification, not on the sampled candidate.

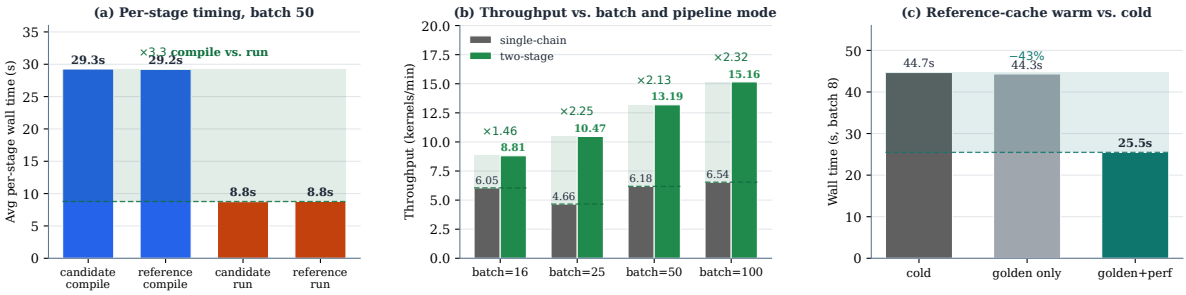


Figure 7: **Sandbox throughput (3 panels).** Legacy KernelBench-25 replicated to batches $\{16, 25, 50, 100\}$ on 108 CPU / 8 MI300X. (a) Per-stage timing (batch 50): compile dominates by $\sim 3.3\times$ (both candidate and reference paths). (b) Throughput vs. batch: two-stage (green) overlaps compile-of-next with run-of-prev, beating single-chain (grey) at every batch. (c) Reference-cache warm vs. cold (batch 8): golden+perf cuts wall time by -43%.

Reusing them removes repeated deterministic work while leaving candidate compile, run, match, and timing checks intact. This distinction matters for RL: the cache improves sample throughput, but it does not create a shortcut around the executable reward contract that the policy is trained against.

6 Related Work

Agentic RL on executable rewards. Recent work trains or scales policies with executable kernel-performance rewards on CUDA or Triton: Dr. Kernel on robust Triton environments and reward leakage (Liu et al., 2026); CUDA-L1 with contrastive reinforcement learning and performance-tool reasoning for CUDA (Li et al., 2026; Nichols et al., 2025); Kevin on multi-turn CUDA RL (Baronio et al., 2025); AutoTriton on Triton RL (Li et al., 2025c); and CuAsmRL on CUDA assembly (He and Yoneki, 2025). HiPO is the first to run an analogous recipe with full-file actions and executable rewards on native HIP, against a GPT-5 best-of-5 HIP reference; the optimisation backbone (PPO, GRPO, DAPO-style asymmetric clipping (Schulman et al., 2017; Shao et al., 2024; Yu et al., 2025)) is standard, the HIP-specific contribution is the playground (§2) and the action contract (§3.1).

HIP / AMD / non-CUDA gap. The closest non-CUDA reference points are Geak (Wang et al., 2025a) and its v2 family, an agent framework (not RL-trained) for Triton on AMD that adds multi-offspring evolution, an LLM-based evaluator, and a hardware-aware feedback loop, with a GEAK-HIP sibling extending the family to native HIP (Wang et al., 2025b). Adjacent non-CUDA work spans SwizzlePerf (Tschand et al., 2025), NPUEval (Kalade and Schelle, 2025), KernelCraft (Nie et al., 2026), and MultiKernelBench (Wen et al., 2025); none target HIP at the full-translation-unit RL-training level. The extended taxonomy (benchmarks, search-based, multi-agent profiling pipelines, domain-specific kernel models) is in App. A.

7 Conclusion

HiPO contributes a high-throughput playground for HIP executable feedback and an entropy-shaped GRPO policy trained on it, locked as HiPO-8B. The playground is the hinge: one async compile/run/match evaluator serves online reward, in-rollout tool calls, and offline KernelBench-HIP scoring, keeping optimisation and evidence on the same executable contract. Its systems evidence is material: compile costs $\mathbf{3.3}\times$ more than run at batch 50, the async pipeline lifts batch-100 wall-time by $\mathbf{2.32}\times$, and warm caching cuts reference-cache time by $\mathbf{43}\%$. Under a GPT-5 best-of-5 reference on KernelBench-HIP (246 kernels, Softmax excluded), HiPO clears GPT-5 parity on **99** kernels at $n{=}1$ vs. 38 for the Qwen3-8B base ($\Delta = +\mathbf{61}$); the F@1.05 lead opens with rollout (**41** vs. 36 at $n{=}8$, the strongest cell), and the Top-10 selector-A geomean reaches $\mathbf{1.461}\times$ vs. $1.257\times$ ($\Delta{=}+\mathbf{0.204}$, $10/10$ at F@1.05; Fig. 1). A shared-memory–tiled `82_conv_depthwise_2D_square` rewrite at $\mathbf{1.35}\times$ illustrates the mechanism on an audited kernel.

To our knowledge this is the first published RL policy to beat a frontier-LLM reference on a public HIP kernel benchmark, which shows that AMD HIP is within reach of executable-reward RL post-training and makes the playground a practical base for AMD-specific curation and reward shaping.

Limitations

AMD-specific idiom coverage. HiPO-8B systematically uses `__shared__` tiling and `__restrict__` pointer hints. AMD-specific idioms (wavefront-aware block sizes, warp-shuffle reductions, vectorised `float4` loads, `#pragma unroll`) appear far less often in our top wins, and we read this as a coverage gap rather than as evidence against the recipe. We frame AMD-specific idiom curation as an open research opportunity for the community, both as a data-curation target (an HIP training corpus that explicitly emphasises these idioms) and as a reward-shaping target on top of the playground (idiom-aware shaping that rewards architecturally appropriate formulations beyond raw speedup).

Surface-similarity copy detector. The DTW copy detector operates on tokenised source and is therefore a surface-similarity proxy. It can flag superficial paraphrases but will, in principle, miss semantic-equivalence rewrites that reorder unrelated tokens or rename non-critical identifiers. We mitigate this with the adaptive length threshold and a sequential cascade that applies the detector only to already-correct candidates, and we deliberately do not report any aggregate that depends on post-hoc copy filtering alone. Building a fully semantic copy detector (AST- or embedding-based) on top of the playground's copy-cascade is an open research opportunity for the community; our DTW surface detector is deliberately a conservative lower bound and is meant to be replaced rather than tuned.

Ethical Considerations

Distillation provenance and license compliance. The HIP reference c_0 is produced by best-of-5 generation from a frontier commercial LLM. We use the resulting code only as an executable-reward target inside our own playground and ship neither verbatim model outputs nor the reference corpus. We follow the provider's usage terms for derived artefacts.

Compute and energy footprint. HiPO-8B training was carried out on a single 108-CPU / 8-MI300X node; per-step wall-clock and full hyperparameters are reported in App. D and App. E. We treat this footprint as non-trivial and report it at step-level rather than as a headline aggregate so that readers can compute a realistic energy bound from the AMD MI300X published TDP. We do not run additional training-seed sweeps in the present paper precisely to keep the footprint bounded.

Hardware diversity and the AMD/CUDA gap. A practical motivation for this work is that public RL kernel recipes concentrate on NVIDIA CUDA. Concentrating optimisation expertise on a single vendor stack is a research-community risk; broadening to HIP and other non-CUDA targets such as NPUs and emerging accelerators (Kalade and Schelle, 2025; Nie et al., 2026) reduces that risk and is part of why we release the playground rather than only the trained checkpoint.

Misuse and dual-use considerations. The artefact is a domain-specific optimiser for HIP GPU kernels. We do not see a direct misuse vector beyond standard concerns about LLM-generated code, which our cascade explicitly mitigates by gating gradient on executable validity rather than on prose plausibility.

References

Martin Andrews and Sam Witteveen. 2025. GPU Kernel Scientist: An LLM-Driven Framework for Iterative Kernel Optimization. *Preprint*, arXiv:2506.20807.

Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. 2025. Kevin: Multi-Turn RL for Generating CUDA Kernels. *Preprint*, arXiv:2507.11948.

Arijit Bhattacharjee, Heng Ping, Son Vu Le, Paul Bogdan, Nesreen K. Ahmed, and Ali Jannesari. 2026. OptiML: An End-to-End Framework for Program Synthesis and CUDA Kernel Optimization. *Preprint*, arXiv:2602.12305.

ByteDance Seed. 2025. CudaLLM: A Language Model for High-Performance CUDA Kernel Generation. Hugging Face model card.

Shiyi Cao, Ziming Mao, Joseph E. Gonzalez, and Ion Stoica. 2026. K-Search: LLM Kernel Generation via Co-Evolving Intrinsic World Model. *Preprint*, arXiv:2602.19128.

Jinwu Chen, Qidie Wu, Bin Li, Lin Ma, Xin Si, Yang Hu, Shouyi Yin, and Jun Yang. 2025. cuPilot: A Strategy-Coordinated Multi-agent Framework for CUDA Kernel Evolution. *Preprint*, arXiv:2512.16465.

Kris Shengjun Dong, Sahil Modi, Dima Nikiforov, Sana Damani, Edward Lin, Siva Kumar Sastry Hari, and Christos Kozyrakis. 2026. KernelBlaster: Continual Cross-Task CUDA Optimization via Memory-Augmented In-Context Reinforcement Learning. *Preprint*, arXiv:2602.14293.

Zacharias V. Fisches, Sahan Paliskara, Simon Guo, Alex Zhang, Joe Spisak, Chris Cummins, Hugh Leather, Gabriel Synnaeve, Joe Isaacson, Aram Markosyan, and Mark Saroufim. 2025. KernelLLM: Making Kernel Development More Accessible. Hugging Face model card.

Junfeng Gong, Zhiyi Wei, Junying Chen, Cheng Liu, and Huawei Li. 2025. From Large to Small: Transferring CUDA Optimization Expertise via Reasoning Graph. *Preprint*, arXiv:2510.19873.

Ping Guo, Chenyu Zhu, Siyuan Chen, Fei Liu, Xi Lin, Zhichao Lu, and Qingfu Zhang. 2025. EvoEngineer: Mastering Automated CUDA Kernel Code Evolution with Large Language Models. *Preprint*, arXiv:2510.03760.

Yuxuan Han, Meng-Hao Guo, Zhengning Liu, Wenguang Chen, and Shi-Min Hu. 2026. Making LLMs Optimize Multi-Scenario CUDA Kernels Like Experts. *Preprint*, arXiv:2603.07169.

Guoliang He and Eiko Yoneki. 2025. CuAsmRL: Reinforcement Learning over CUDA Assembly. In *ACM/IEEE International Symposium on Code Generation and Optimization*.

Sarunas Kalade and Graham Schelle. 2025. NPUEval: Optimizing NPU Kernels with LLMs and Open Source Compilers. *Preprint*, arXiv:2507.14403.

Robert Tjarko Lange, Qi Sun, Aaditya Prasad, Maxence Faldor, Yujin Tang, and David Ha. 2025. Towards Robust Agentic CUDA Kernel Benchmarking, Verification, and Optimization. *Preprint*, arXiv:2509.14279.

Kelun Lei, Hailong Yang, Huaitao Zhang, Xin You, Kaige Zhang, Zhongzhi Luan, Yi Liu, and Depei Qian. 2025. PRAGMA: A Profiling-Reasoned Multi-Agent Framework for Automatic Kernel Optimization. *Preprint*, arXiv:2511.06345.

Haonan Li, Keyu Man, Partha Kanuparthy, Hanning Chen, Wei Sun, Sreen Tallam, Chenguang Zhu, Kevin Zhu, and Zhiyun Qian. 2025a. TritonForge: Profiling-Guided Framework for Automated Triton Kernel Optimization. *Preprint*, arXiv:2512.09196.

Jianling Li, ShangZhan Li, Zhenye Gao, and 1 others. 2025b. TritonBench: Benchmarking Large Language Model Capabilities for Generating Triton Operators. In *Findings of the Association for Computational Linguistics*.

Shangzhan Li, Zefan Wang, Ye He, Yuxuan Li, Qi Shi, Jianling Li, Yonggang Hu, Wanxiang Che, Xu Han, Zhiyuan Liu, and Maosong Sun. 2025c. AutoTriton: Automatic Triton Programming with Reinforcement Learning in LLMs. *Preprint*, arXiv:2507.05687.

Xiaoya Li, Xiaofei Sun, Albert Wang, Jiwei Li, and Chris Shum. 2026. CUDA-L1: Improving CUDA Optimization via Contrastive Reinforcement Learning. *Preprint*, arXiv:2507.14111.

Edward Lin, Sahil Modi, Siva Kumar Sastry Hari, and 1 others. 2026. SOL-ExecBench: Speed-of-Light Benchmarking for Real-World GPU Kernels Against Hardware Limits. *Preprint*, arXiv:2603.19173.

Wei Liu, Jiawei Xu, Yingru Li, Longtao Zheng, Tianjian Li, Qian Liu, and Junxian He. 2026. Dr. Kernel: Reinforcement Learning Done Right for Triton Kernel Generations. *Preprint*, arXiv:2602.05885.

Meta PyTorch Team. 2025. BackendBench: Ship Correct and Fast LLM Kernels to PyTorch. GitHub repository.

Daniel Nichols, Konstantinos Parasyris, Charles Jekel, Abhinav Bhatele, and Harshitha Menon. 2025. Integrating Performance Tools in Model Reasoning for GPU Kernel Optimization. *Preprint*, arXiv:2510.17158.

Jiayi Nie, Haoran Wu, Yao Lai, Zeyu Cao, Cheng Zhang, Binglei Lou, Erwei Wang, Jianyi Cheng, Timothy M. Jones, Robert Mullins, Rika Antonova, and Yiren Zhao. 2026. KernelCraft: Benchmarking for Agentic Close-to-Metal Kernel Generation on Emerging Hardware. *Preprint*, arXiv:2603.08721.

NVIDIA. 2025. ComputeEval: A Framework for Evaluating LLM Code Generation on GPU Compute Kernels. GitHub repository.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. 2025. KernelBench: Can LLMs Write Efficient GPU Kernels? *Preprint*, arXiv:2502.10517.

Sahan Paliskara and Mark Saroufim. 2025. KernelBook: A Torch-to-Triton Kernel Translation Dataset. Hugging Face dataset.

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. 2025. KernelBand: Steering LLM-Based Kernel Optimization via Hardware-Aware Multi-Armed Bandits. *Preprint*, arXiv:2511.18868.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. *Preprint*, arXiv:1707.06347.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models. *Preprint*, arXiv:2402.03300.

Arya Tschand, Muhammad Awad, Ryan Swann, Kesavan Ramakrishnan, Jeffrey Ma, Keith Lowery, Ganesh Dasika, and Vijay Janapa Reddi. 2025. SwizzlePerf: Hardware-Aware LLMs for GPU Kernel Performance Optimization. *Preprint*, arXiv:2508.20258.

Jianghui Wang, Vinay Joshi, Saptarshi Majumder, Chao Xu, Bin Ding, Ziqiong Liu, Pratik Prabhanjan Brahma, Dong Li, Zicheng Liu, and Emad Barsoum. 2025a. GEAK: Introducing Triton Kernel AI Agent and Evaluation Benchmarks. *Preprint*, arXiv:2507.23194.

Jianghui Wang, Saptarshi Majumder, Umang Pandey, Chao Xu, Bin Ding, Manoj Rao, Ziqiong Liu, Pratik Prabhanjan Brahma, Dong Li, Zicheng Liu, and Emad Barsoum. 2025b. GEAK-Triton v2 Family of AI Agents: Kernel Optimization for AMD Instinct GPUs. ROCm Blogs. Includes companion GEAK-HIP for native HIP code optimization.

Anjiang Wei, Tianran Sun, Yogesh Seenichamy, Hang Song, Anne Ouyang, Azalia Mirhoseini, Ke Wang, and Alex Aiken. 2025. Astra: A Multi-Agent System for GPU Kernel Performance Optimization. *Preprint*, arXiv:2509.07506.

Zhongzhen Wen, Yinghui Zhang, Zhong Li, Zhongxin Liu, Linna Xie, and Tian Zhang. 2025. MultiKernelBench: A Multi-Platform Benchmark for Kernel Generation. *Preprint*, arXiv:2507.17773.

Nina Wiedemann, Quentin Leboutet, Michael Paulitsch, Diana Wofk, and Benjamin Ummenhofer. 2026. KernelFoundry: Hardware-Aware Evolutionary GPU Kernel Optimization. *Preprint*, arXiv:2603.12440.

Qiying Yu, Chenyuan Wu, Zheng Zhang, Yufan Zhang, Hao Liu, Yuxuan Tong, Xiangpeng Wei, Yonghui Wu, and Mingxuan Wang. 2025. DAPO: An Open-Source LLM Reinforcement Learning System at Scale. *Preprint*, arXiv:2503.14476.

Zijian Zhang, Rong Wang, Shiyang Li, Yuebo Luo, Mingyi Hong, and Caiwen Ding. 2025. CudaForge: An Agent Framework with Hardware Feedback for CUDA Kernel Optimization. *Preprint*, arXiv:2511.01884.

Appendix Contents

- **App. A** – Extended related work
- **App. B** – Token-level DTW copy detector
- **App. C** – Training dynamics (entropy / reward / grad-norm)
- **App. D** – Reproducibility
- **App. E** – Training configuration
- **App. F** – HiPO-8B training algorithm
- **App. G** – Additional kernel case studies
- **App. H** – GRPO update: full derivation
- **App. I** – Metric detail: selectors A and B

A Extended Related Work

The body Related Work (§6) keeps only HiPO's home camp (agentic RL on executable rewards) and the HIP / AMD / non-CUDA gap. This appendix gives the full taxonomy across the remaining four directions: benchmarks and measurement, the executable-reward macro-trend, search-based kernel optimisation, multi-agent profiling-and-edit pipelines, and domain-specific kernel models.

Benchmarks and measurement. KernelBench (Ouyang et al., 2025) established the PyTorch-to-CUDA kernel-generation setting that we extend to PyTorch-to-HIP, alongside the PyTorch-to-Triton line (Paliskara and Saroufim, 2025; Fisches et al., 2025) and the PyTorch-op-level evaluation suites (Meta PyTorch Team, 2025; ByteDance Seed, 2025). Complementary suites cover Triton operators (Li et al., 2025b), multi-platform backends (Wen et al., 2025), verifier-hardened agentic benchmarking (Lange et al., 2025; Lin et al., 2026), NVIDIA ComputeEval (NVIDIA, 2025), Meta BackendBench (Meta PyTorch Team, 2025), and emerging-hardware benches such as KernelCraft (Nie et al., 2026) and AMD NPUEval (Kalade and Schelle, 2025). Robust-kbench-style audits show verifier holes can dominate reported speedups, so we exclude `23_Softmax_base`. HIP and other non-CUDA stacks remain under-served relative to CUDA/Triton.

Executable-reward kernel RL as an emerging paradigm. Two macro-trends frame this work. Supervision has shifted from human-preference signals to executable rewards, exemplified by CUDA-L1's contrastive RL (Li et al., 2026) and Dr. Kernel's reward-leakage controls on Triton (Liu et al., 2026). Abstraction is moving up the stack: kernel- and runtime-generation pipelines couple LLMs with profiling-and-edit loops (Wei et al., 2025; Chen et al., 2025; Lei et al., 2025; Andrews and Witteveen, 2025), with K-Search (Cao et al., 2026) framing optimisation as world-model co-evolution and KernelBlaster / Multi-Scenario (Dong et al., 2026; Han et al., 2026) pushing evolutionary orchestration; KernelBench supplies the measurement ground-truth (Ouyang et al., 2025). The combined paradigm is in principle hardware-portable, yet published recipes are almost exclusively CUDA or Triton; AMD HIP is the largest open GPU stack with no public kernel-RL baseline. The kernel-LLM literature partitions into four directions treated in turn below (search-based, agent-based, domain-specific, agentic-RL), and HiPO sits in the agentic-RL axis distinguished on three coordinates: a single-agent React loop versus multi-agent orchestration; a full-file translation-unit action versus snippet patches; and the HIP substrate versus CUDA/Triton.

Search-based kernel optimisation. This camp treats the LLM as a black-box generator inside an outer-loop search. Representative systems are K-Search (Cao et al., 2026), EVOENGINEER (Guo et al., 2025), KernelFoundry (Wiedemann et al., 2026), OptiML (Bhattacharjee et al., 2026), KernelBand (Ran et al., 2025), TritonForge (Li et al., 2025a), From-Large-to-Small CUDA transfer (Gong et al., 2025), and CudaForge's hardware-feedback multi-agent search (Zhang et al., 2025). The optimisation outer loop is the contribution and the underlying LLM is held fixed. HiPO is orthogonal: a single agent is trained by RL on execution rewards, and the sandbox is the evaluator, not the search loop.

Agent-based kernel pipelines. This camp pairs generation with profiling-and-edit multi-agent loops on CUDA: Astra, cuPilot, PRAGMA, GPU Kernel Scientist, KernelBlaster, and Multi-Scenario (Wei et al., 2025; Chen et al., 2025; Lei et al., 2025; Andrews and Witteveen, 2025; Dong et al., 2026; Han et al., 2026). The agent or search loop is the contribution; the base model is usually held fixed. HiPO inverts this: orchestration is deliberately minimal (single-agent React, 3-turn refine), and the contribution is the trained policy distribution.

Domain-specific kernel models. A parallel line trains domain-specific code models on hand-curated kernel corpora: Meta KernelLLM (Fisches et al., 2025), ByteDance CudaLLM (ByteDance Seed, 2025), and the GPUMODE Kernel-Book (Paliskara and Saroufim, 2025) Torch-to-Triton dataset they build on; further points are surveyed in Ouyang et al. (2025). HiPO's recipe is base-model agnostic and is meant to compose with such models rather than compete with them; the HiPO-8B checkpoint is one realisation on Qwen3-8B.

B Token-level DTW copy detector

The copy gate $C(c)$ in Eq. 1 fires on a normalised token-level DTW distance to the reference c_0. The body of the paper gives the working range $d_{\text{copy}}(L_c) \in [0.05, 0.15]$ but not the operator; we make it precise here.

Pipeline. Given candidate c and reference c_0, we (i) extract the body of the `__global__` kernel from each file, (ii) tokenise the body via a regex lexer (identifiers, operators, numeric and string literals; line and block comments stripped), and (iii) compute a token-level DTW distance with unit substitution cost. The detector runs only on candidates that already satisfy executable validity $V(c)$ (Eq. 1), so it disambiguates among already-correct candidates rather than gating compile/run/match.

Tokenisation. Let $\tau(\cdot)$ denote the lexer output, with $\tau_1{=}\tau(\text{body}(c_0))$, $\tau_2{=}\tau(\text{body}(c))$, lengths $L_i{=}|\tau_i|$, and $L_c{=}\max(L_1, L_2)$.

DTW recurrence. With unit substitution cost $\delta(a, b){=}\mathbf{1}[a{\neq}b]$, define

$$\begin{aligned} D[i,j] \;&=\; \delta(\tau_1[i], \tau_2[j]) \\ &+ \min\big(D[i{-}1, j{-}1],\, D[i{-}1, j],\, D[i, j{-}1]\big), \end{aligned}$$

with $D[0, 0] = 0$ and $D[i, 0] = D[0, j] = \infty$ for $i, j \geq 1$.

Normalised distance.

$$d_{\text{DTW}}(c, c_0) \;=\; \frac{D[L_1, L_2]}{\max(L_1, L_2)} \;\in\; [0, 1]. \qquad (4)$$

Normalisation by $\max(L_1, L_2)$ keeps the distance comparable across the wide HIP kernel-length distribution.

Adaptive copy threshold. The threshold $d_{\text{copy}}(L_c)$ is parameterised in an absolute edited-token count $k_{\text{copy}}(L_c)$ and clipped to the working range $[0.05, 0.15]$: $d_{\text{copy}}(L_c) = k_{\text{copy}}(L_c)/L_c$, with $k_{\text{copy}}(L_c) = 6$ for $L_c < 50$ (short kernels need fewer absolute edits to count as novel), $k_{\text{copy}}(L_c) = 8$ for $50 \leq L_c \leq 200$ (standard band, centred at $L_c = 100 \rightarrow d_{\text{copy}} = 0.08$), and $k_{\text{copy}}(L_c){=}\min(8\,L_c/100, 30)$ for $L_c{>}200$ (long kernels scale proportionally up to a 30-token cap). The copy gate fires iff $d_{\text{DTW}}(c, c_0){<}d_{\text{copy}}(L_c)$.

Why token-DTW and not edit distance. DTW allows one-to-many token alignments and so absorbs short insertions and deletions (e.g. added `__restrict__` qualifiers or a swapped block-size constant) more gracefully than a strict edit distance, which would punish length-changing rewrites that preserve the underlying computation. The trade-off is that token-DTW is still a *surface*-similarity proxy: it cannot detect semantic-equivalence rewrites that reorder unrelated tokens or rename non-critical identifiers (see §7). The cascade structure of Eq. 1 (copy detection *after* match) bounds the impact of this proxy to the disambiguation step among already-correct candidates.

C Training Dynamics (entropy / reward / grad-norm)

Fig. 8 complements the per-prompt reward curve in §3 (Fig. 4) with the two stability diagnostics that determined the locked HiPO-8B checkpoint. Policy entropy $\overline{H}_\theta$ holds in a narrow mid-band (EMA $0.35 \rightarrow 0.37$ nats), never collapsing to a near-deterministic regime under the detached entropy-shaped advantage (Eq. 2, $\alpha{=}0.4$, $\kappa{=}2$) combined with a small PPO entropy coefficient. Per-prompt mean reward $\bar{R}(x)$ climbs steadily out of the cold-start regime (EMA $\sim 0.14 \rightarrow 0.56$) and stabilises with no late-training saturation. Policy-gradient norm $\|\nabla\|$ stays in a narrow band (EMA $\sim 0.16 \rightarrow 0.21$) consistent with the asymmetric clip cap (Eq. 3, $(c_{\text{lo}}, c_{\text{hi}}){=}(0.8, 1.4)$) and the constant-$T_{\max}$ aggregator (Eq. 6); no detected divergence or instability spike up to the locked checkpoint. Together: entropy holds, reward holds, gradient holds; the three empirical preconditions for stopping training and locking the checkpoint used throughout this paper.

Figure 8: **Training dynamics up to the locked HiPO-8B checkpoint (3 panels).** (a) Policy entropy $\overline{H}_\theta$ (EMA-smoothed); (b) per-prompt mean reward $\bar{R}(x)$ (EMA); (c) policy-gradient norm $\|\nabla\|$ (EMA). Series are truncated at the locked checkpoint, marked in HiPOGreen.

D Reproducibility

Apples-to-apples inference setup. Both the Qwen3-8B baseline and HiPO-8B are evaluated under a single orchestrator with identical React multi-turn template, identical decoding parameters, identical rollout schedule ($n \in \{1, 4, 16\} \times 3$ turns), and identical sandbox configuration; the two runs differ only in the model checkpoint and the output directory. All training and evaluation code, configuration files, and analysis scripts will be released as a single repository upon de-anonymisation.

Aggregation procedure. Tab. 1 (primary per-level pair-block) is computed from per-kernel speedups produced by the unified evaluator. `23_Softmax_base` is excluded *before* ranking and aggregation. Geomean uses the paired missing-cell rule missing $\rightarrow 1.0\times$ so the baseline and HiPO denominators match on every (level, n) cell. Tab. 2 and Tab. 4 re-aggregate the same per-kernel data under top-k selectors A (top-k by HiPO) and B (top-k by $\max$), $k \in \{10, 20, 30\}$.

Sandbox-throughput measurements. The numbers in Fig. 7 (single-chain vs. two-stage, per-stage timing, warm-vs-cold cache) are measured on the same evaluator core that produces Tab. 1, on a 108-CPU / 8-GPU MI300X host; reported values are wall-time medians over five runs at each batch size. The implementation entry points for the design pillars of §2 are `sandbox_core/runtime.py` (per-GPU `asyncio.Semaphore(1)` runtime pool), `sandbox_core/reference.py` (the three-layer cache and its key schema), and `server_adapters/master.py` (the local-plus-remote round-robin dispatcher).

Training reproducibility. The reward curve in Fig. 4 is the per-prompt mean reward exported from the experiment-tracking dashboard for the HiPO-8B training session (200 steps, mean/min/max channels). The locked checkpoint of this run is the model evaluated everywhere in this paper. Full hyperparameters: App. E.

E Training Configuration

Tab. 3 consolidates every hyperparameter used to train HiPO-8B, grouped by role. The values are not the contribution; the rationale rows beneath the table give the design intent for the few non-obvious choices.

Rationale for non-obvious choices. *Asymmetric clip* $(0.8, 1.4)$. A symmetric clip (ϵ=0.2 both sides) treats large positive and negative advantage updates equally; under sparse executable rewards, the rare large positive (a kernel that finally beats reference) is exactly what we want the policy to commit to. Widening the upper bound to 1.4 (ϵ_{hi}=0.4) keeps PPO trust-region behaviour while explicitly biasing exploration upside. *Entropy detached, capped.* An undetached entropy bonus backpropagates a maximise-entropy gradient that competes with the reward gradient; detaching the entropy estimate and capping its contribution at $|\hat{A}|/2$ makes the bonus a soft floor that can only ever *add* to the magnitude of an existing reward signal, never flip its sign. *Length-invariant aggregator.* Per-batch mean-length normalisation makes a long-response batch effectively scale the gradient down; on full-file HIP responses the length distribution is wide ($\sim$1K–16K tokens), so we instead divide by the constant $T_{\max}$, keeping the gradient scale comparable across rollout shapes. *Copy suppression in reward, not loss.* Penalising near-copy candidates via r_{copy}=0 keeps the policy gradient minimal (no auxiliary KL term) and avoids the well-known KL/copy gradient interference seen when behavioural cloning of the reference is implicit through KL.

F HiPO-8B training algorithm

The full HiPO-8B training loop with the gated reward cascade, the group-normalised entropy-shaped advantage, and the asymmetric-clip surrogate written out per line is Alg. 1 below. The prose paragraphs after the algorithm give the verbatim expansion of each highlighted step.

Highlight 1: gated reward (pointer). Eq. 1 and the symbol gloss in §3 (*Reward terms*) already specify the cascade, the adaptive copy threshold $d_{\text{copy}}(L_c) \in [0.05, 0.15]$, and the perf-branch shaping $\min(s, s_{\max}) + b_+ \cdot \mathbf{1}[s > 1.1]$ with $s_{\max} = 10$,

Hyperparameter	Value
Optimisation	
Base model	Qwen3-8B
Learning rate η	1×10^{-6}
Optimiser	AdamW (default β, weight decay 0)
Train batch size $\lvert\mathcal{B}\rvert$	8 prompts
GRPO group size G	8 rollouts / prompt
Max response length $T_{\max}$	16K tokens
Prompt length budget	8K tokens
Decoding (rollouts)	
Temperature	1.0
Top-p	0.95
Inference engine	in-actor vLLM
Loss	
Clip range $(c_{\text{lo}}, c_{\text{hi}})$	$(0.8, 1.4)$ asymmetric
Equivalent $(\epsilon_{\text{lo}}, \epsilon_{\text{hi}})$	$(0.2, 0.4)$
Advantage shaping	detached entropy add (Eq. 2)
Entropy mix α	0.4
Entropy cap divisor κ	2
Entropy bonus β_{ent}	0.02
KL penalty β_{KL}	0
Aggregation	seq-mean-token-sum-norm (Eq. 6)
Aggregator divisor	constant $T_{\max}$
Reward	
Mode	correctness-gated, copy-suppressed (Eq. 1)
r_{ok}	0.3
$s_{\max}$ (speedup clip)	$10\times$
r_{copy}	0
DTW copy threshold $d_{\text{copy}}(L_c)$	adaptive in token length, default range $[0.05,\ 0.15]$
Speedup bonus $b(c)$	active for $s > 1.1$
Rollout	
Turns (training)	1 (single-turn React)
Turns (inference eval)	3 (multi-turn React; §4)
Inter-turn state (eval only)	verdict + compact CoT summary
Action contract	strict JSON {thought, code}
Code requirement	complete HIP translation unit
Data filtering	
Dynamic-group filtering	on accuracy variance within group
Verifier exclusion	`23_Softmax_base`

Table 3: **HiPO-8B training configuration.** All values are verbatim from the locked training run of §3.

$r_{\text{ok}} = 0.3$, $r_{\text{copy}} = 0$, $b_+ = 0.3$. We highlight only that the cascade is sequential by construction, so copy detection disambiguates correct candidates and never contaminates the gradient with rewards on broken kernels.

Highlight 2: group-normalised, entropy-shaped advantage. With G rollouts $\{y^{(i)}\}_{i=1}^{G}$ for prompt x_b and per-rollout reward $R^{(i)}$, define

$$\mu_{x_b} = \frac{1}{G}\sum_i R^{(i)}, \quad \sigma^2_{x_b} = \frac{1}{G}\sum_i (R^{(i)} - \mu_{x_b})^2 + \varepsilon.$$

The group-normalised advantage is $\hat{A}^{(i)} = (R^{(i)} - \mu_{x_b})/\sigma_{x_b}$ (Eq. 5); the entropy-shaped advantage adds a detached, capped entropy bonus to its absolute value:

$$\tilde{A}_t^{(i)} = \hat{A}^{(i)} + \min\Big(\alpha \cdot \text{sg}\big[H_\theta(s_t^{(i)})\big],\ |\hat{A}^{(i)}|/\kappa\Big).$$

$\text{sg}[\cdot]$ blocks gradient flow through the entropy term, so the bonus contributes only to the magnitude of an already-signed advantage; the cap $|\hat{A}^{(i)}|/\kappa$ prevents the bonus from overwhelming the reward sign on rollouts whose advantage is small.

Algorithm 1 HiPO-8B training loop.

Require: policy π_θ; HIP corpus $\mathcal{D}$; group size G; batch size $|\mathcal{B}|$; max length $T_{\max}$; clip $(c_{\text{lo}}, c_{\text{hi}})$; entropy params $(\alpha, \kappa, \beta_{\text{ent}})$; reward params $(r_{\text{ok}}, s_{\max}, r_{\text{copy}}, b_+)$; learning rate η

for each training step **do**
  Sample prompt batch $\{x_b\}_{b=1}^{|\mathcal{B}|} \sim \mathcal{D}$; $\theta_{\text{old}} \leftarrow \theta$
  for each prompt x_b **do**
    Rollout: single-turn React (one JSON `thought`/`code` message per candidate).
    Sample G candidates $\{y^{(i)}\} \sim \pi_{\theta_{\text{old}}}(\cdot|x_b)$
    Playground returns $(\text{compile}, \text{run}, \text{match}, s, C)^{(i)}$
    Reward: gated cascade, per candidate (Eq. 1).
      $R^{(i)} = 0$ **if** $\neg V(c)$
      $R^{(i)} = r_{\text{copy}}$ **if** $V(c) \wedge C(c)$
      $R^{(i)} = r_{\text{ok}} + \min(s, s_{\max}) + b(c)$ **if** $V(c) \wedge \neg C(c)$
    Advantage: group-norm + entropy shape (Eq. 5–2).
      $\hat{A}^{(i)} = (R^{(i)} - \mu_{x_b})/\sigma_{x_b}$
      $\tilde{A}_t^{(i)} = \hat{A}^{(i)} + \min\big(\alpha\, \text{sg}[H_\theta],\ |\hat{A}^{(i)}|/\kappa\big)$
  Loss: asymmetric clip + length-invariant agg. (Eq. 3–6).
    $\mathcal{L}_t^{(i)} = -\min\big(u_t^{(i)} \tilde{A}_t^{(i)},\ \text{clip}(u_t^{(i)}, c_{\text{lo}}, c_{\text{hi}}) \tilde{A}_t^{(i)}\big)$
    $\mathcal{L}_{\text{batch}} = \frac{1}{T_{\max}} \sum_{i,t} m_t^{(i)}\, \mathcal{L}_t^{(i)}$
  $\theta \leftarrow \theta - \eta\, \nabla_\theta \big(-\mathcal{L}_{\text{batch}} + \beta_{\text{ent}} \overline{H}_\theta\big)$

Highlight 3: asymmetric clip and length-invariant aggregation. The token-level surrogate uses an asymmetric clip $(c_{\text{lo}}, c_{\text{hi}}) = (0.8, 1.4)$, biased toward upside exploration:

$$\mathcal{L}_t^{(i)} = -\min\big(u_t^{(i)}\, \tilde{A}_t^{(i)},\ \text{clip}(u_t^{(i)}, c_{\text{lo}}, c_{\text{hi}})\, \tilde{A}_t^{(i)}\big).$$

Aggregation divides by the constant $T_{\max}$ (not per-batch mean length), keeping gradient scale comparable across the wide HIP response-length distribution: $\mathcal{L}_{\text{batch}} = \frac{1}{T_{\max}} \sum_{(x_b, y^{(i)}) \in \mathcal{B}} \sum_{t=1}^{T_i} m_t^{(i)} \mathcal{L}_t^{(i)}$.

Rollout scope. The training rollout is single-turn: each $y^{(i)}$ is one JSON `thought`/`code` message sampled from $\pi_{\theta_{\text{old}}}(\cdot \mid x_b)$ and scored end-to-end by the playground. The multi-turn React loop with compact CoT summaries is an inference-time scaling protocol, used only in the KernelBench-HIP evaluation sweep of §4 (Data and protocol), and does not appear in the training gradient.

G Additional Kernel Case Studies

Two kernels carry the most interpretable HiPO-8B wins; both fall out of the action contract of §3.1 (full-file HIP generation with executable-reward gating) and the exploration-shaped GRPO recipe of §3.2. Stylised before/after diagrams of both rewrites are in Fig. 9.

`44_Average_Pooling_1D` (L1, 1.21×). The GPT-5 best-of-5 reference computes the input index with two chained multiplies inside the inner loop and routes every element through a `__shared__` staging tile guarded by an `if (idx < input_length)` check (Fig. 9a, left). HiPO-8B keeps the same algorithm but hoists the flat index `((b_idx*C+c_idx)*I)+i_start` above the pool window, removes the shared-memory hop, and annotates the inner loop with `#pragma unroll`; the resulting kernel is a straight strided reduction that the HIP compiler turns into contiguous global loads (Fig. 9a, right). The win is an indexing/idiom micro-rewrite, not a structural one, but it is the cleanest audited single-kernel win at L1.

`82_conv_depthwise_2D_square` (L1, 1.35×). Both rollouts share the same depthwise structure with `__shared__` staging of the input and filter tiles (Fig. 9b, left). HiPO-8B keeps the filter tile but drops the input-tile staging, lets each thread read its $K \times K$ neighbourhood directly from L1, and widens the block from 16×16 to 32×16 so the wider thread row absorbs the lost shared-memory reuse in registers (Fig. 9b, right). The full-file action permits this register-vs-LDS trade-off and the in-rollout compile/profile feedback stabilises the exploration.

Idiom incidence on the wider top-Δ set. Across the wider HiPO-8B top-Δ set, `__restrict__` and `__shared__` are the most-hit idioms (7× and 5× respectively, with 5 entries combining both). The rewrites suggest that shared-memory tiling with restrict-qualified pointers is becoming a transferable pattern for HiPO-8B. AMD-specific idioms (wavefront-aware block sizes, warp-shuffle reductions, vectorised loads, `#pragma unroll`) remain rare and are a clear data-curation target.

For completeness we include the two audited HiPO-8B rewrites below. Each appendix figure shows the full GPT-5 best-of-5 HIP reference (left column) alongside the HiPO-8B rewrite (right column); HiPO-only edits are highlighted in light green and baseline-only lines in light red, with runs of more than five consecutive identical lines collapsed into a single banner so the entire kernel structure stays visible. Per-kernel speedups are the real measured ratios from the best-of-16 rollout (`best_valid_summary.csv`) against the GPT-

5 reference, not against the unfused PyTorch eager baseline. File paths, generation indices, and rollout markers are scrubbed from the displayed code for anonymity.

Idiom recurrence. Across these two audited-clean cases, HiPO-8B exposes two useful edit modes. `82_conv_depthwise_2D_square` (Fig. 9b) is a structural rewrite: HiPO-8B drops the input-tile `__shared__` stage, widens the block, and lets registers absorb the $K \times K$ reuse. `44_Average_Pooling_1D` (Fig. 9a) is an idiom-level rewrite on an already-strong GPT-5 reference: hoisted flat indexing, removal of an unnecessary shared-memory hop, and `#pragma unroll`. The split points at the next data-curation direction: bias the rollout towards prompts where HiPO has structural room rather than asking it to find micro-deltas on already-converged kernels.

H GRPO update: full derivation

The body retains the two operations whose HIP-specific framing matters most: the gated reward (Eq. 1) and the asymmetric clipped surrogate (Eq. 3). The remaining group-baseline, entropy-shaping, and aggregation steps are standard GRPO/DAPO components and are written out here for reproducibility.

Setup. Prompts $x \sim \mathcal{D}$; group of $G{=}8$ rollouts $\{y^{(i)}\}_{i=1}^{G}$ from a behaviour policy $\pi_{\theta_{\text{old}}}$; rollout length $T_i \le T_{\max} = 16\text{K}$; states $s_t^{(i)} = (x, y_{<t}^{(i)})$; per-rollout reward $R^{(i)} \equiv R(x, y^{(i)})$ from Eq. 1; token-level importance ratio $u_t^{(i)} \equiv \pi_\theta(y_t^{(i)}|s_t^{(i)})/\pi_{\theta_{\text{old}}}(y_t^{(i)}|s_t^{(i)})$.

Group-normalised advantage.

$$\hat{A}_t^{(i)} = \frac{R^{(i)} - \mu_x}{\sigma_x}, \tag{5}$$

with $\mu_x{=}\frac{1}{G}\sum_i R^{(i)}$, $\sigma_x^2{=}\frac{1}{G}\sum_i (R^{(i)}{-}\mu_x)^2{+}\varepsilon$ ($\varepsilon{=}10^{-8}$). The group baseline removes the learned critic at cost of a within-prompt z-score that self-normalises across difficulty.

Length-invariant aggregation and total objective.

$$\mathcal{L}_{\text{batch}}(\theta) = \frac{1}{T_{\max}} \sum_{(x,y^{(i)})\in\mathcal{B}} \sum_{t=1}^{T_i} m_t^{(i)}\, \mathcal{L}_t^{(i)}(\theta), \tag{6}$$

$$\mathcal{J}(\theta) = -\mathcal{L}_{\text{batch}}(\theta) + \beta_{\text{ent}}\, \overline{H}_\theta, \tag{7}$$

with $m_t^{(i)} \in \{0,1\}$ the response mask, $\beta_{\text{ent}}{=}0.02$, and $\beta_{\text{KL}}{=}0$. We divide by the constant $T_{\max}$ rather than per-batch mean length, keeping gradient scale invariant across the wide HIP response-length distribution. Copy suppression lives in the reward (Eq. 1), not the loss, avoiding the well-known KL/copy gradient interference that arises when behavioural cloning of the reference is implicit through KL.

I Metric detail: selectors A and B

The body Tab. 1 reports per-level paired results on all 246 kernels. Tab. 2/4 re-aggregate the same per-kernel data on two complementary top-k subsets ($k \in \{10, 20, 30\}$).

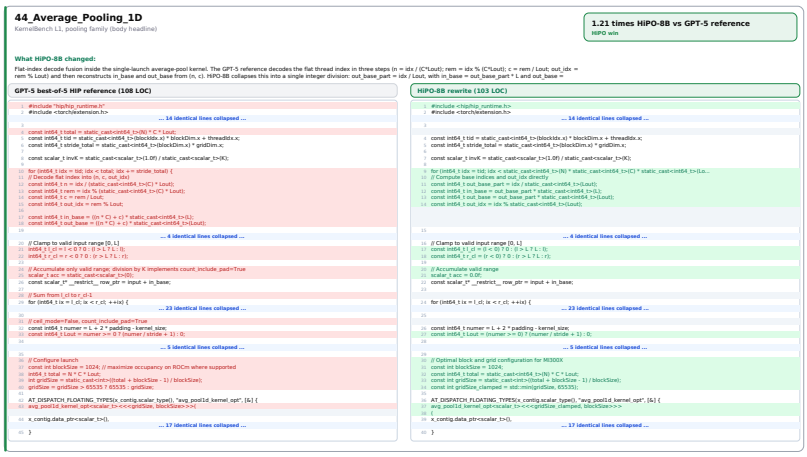
Selector A (top-k by HiPO). Ranks kernels by HiPO-8B best-of-16 speedup, descending, then takes the top-k. Surfaces the kernels on which HiPO is most effective; by construction favourable to HiPO, hence reported alongside selector B.

Selector B (top-k by $\max$). Ranks kernels by $\max(\text{HiPO}, \text{baseline})$ best-of-16 speedup, descending, then takes the top-k. Selects kernels where *either* system finds non-trivial headroom; cannot be inflated by HiPO appearing only on its own favourable picks.

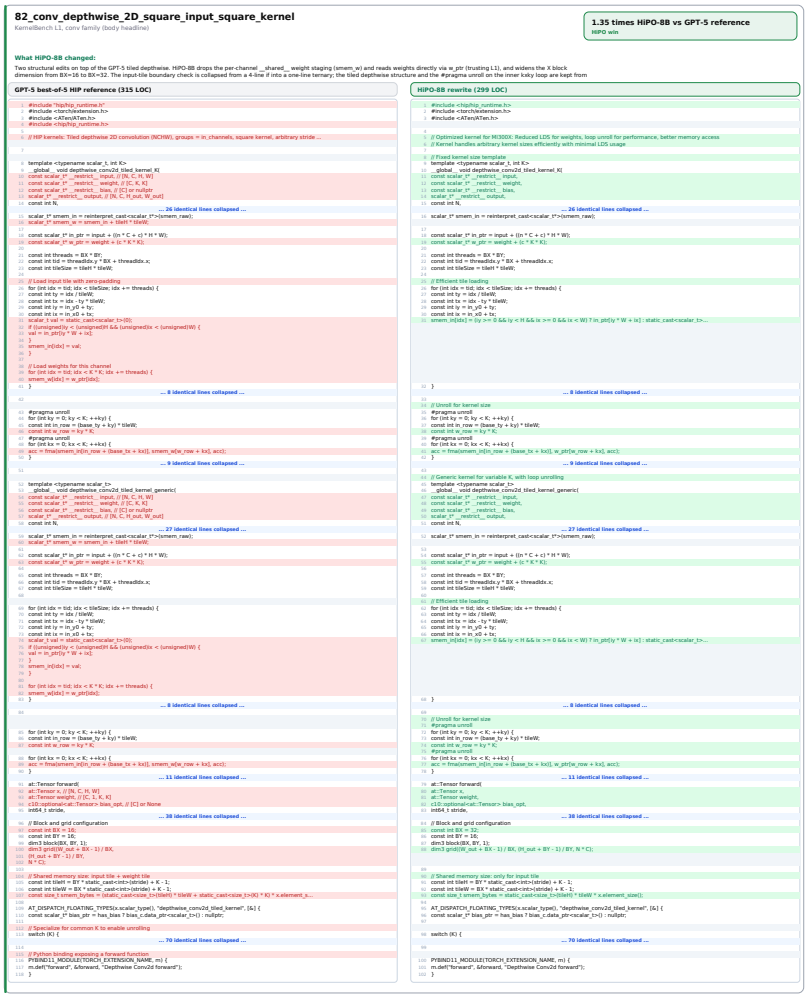
		Top-30 (B)			**Top-20 (B)**			**Top-10 (B)**		
Setup	n	B-geo	H-geo	Δ	B-geo	H-geo	Δ	B-geo	H-geo	Δ
Single-shot	1	0.790	**0.861**	+0.071	0.695	**0.857**	+0.162	0.477	**0.852**	+0.376
Best-of-4	4	1.141	1.069	−0.072	1.178	1.072	−0.106	1.317	1.073	−0.244
Best-of-8	8	1.161	1.128	−0.033	1.216	1.169	−0.047	1.390	1.233	−0.157
Best-of-16	16	1.178	1.139	−0.038	1.236	1.182	−0.054	1.424	1.251	−0.173

Table 4: **Headroom-controlled deep cut (selector B: top-k by $\max$).** Companion to Tab. 2: rows where at least one system shows non-trivial speedup, so the contrast cannot be inflated by HiPO appearing only on its own favourable picks. *B-geo* and *H-geo* carry the same meaning as in Tab. 2 (paired geomean of base / HiPO under the missing-cell rule). At $n{=}1$ this stricter view shows HiPO leading on tighter cuts; at $n{=}16$ HiPO *trails* on geomean because the ranking selects kernels where qwen already had headroom at best-of-16; the expected anti-cherry-pick control, not a contradiction of Tab. 2.

Aggregation rules. Paired geomean uses the missing-cell rule missing $\to 1.0\times$ so baseline and HiPO denominators are identical on every cell. Fast@p is the share of kernels with best-history speedup $\ge p$ (penalised: failures count as 0). W/D/L compares HiPO vs. baseline at $\pm 5\%$ on the $n{=}16$ column: win if HiPO is $> 5\%$ faster,



(a) `44_Average_Pooling_1D` (1.21× at n=16, L1). Baseline computes the input index via repeated multiplies and stages through `__shared__` memory before a guarded inner reduction. HiPO-8B drops the shared-memory hop, hoists the flat index `((b_idx*C+c_idx)*I)+i_start` above the inner loop, and uses `#pragma unroll` on the pool window, leaving an arithmetic pattern the compiler turns into contiguous global loads.



(b) `82_conv_depthwise_2D_square` (1.35× at n=16, L1). Baseline stages a $(BY+K-1) \times (BX+K-1)$ input tile and the $K \times K$ filter into `__shared__` memory, then reads them back in the inner $K \times K$ accumulate. HiPO-8B keeps shared-memory staging only for the filter, doubles the block width (16×16 → 32×16), and lets each thread read its $K \times K$ neighbourhood directly from L1, swapping shared-memory bandwidth for register reuse.

Figure 9: **Two audited HiPO-8B kernel rewrites.** Diff-bearing lines are highlighted (HiPOGreen). Both rewrites are byte-exact on fresh seeds.

draw within $\pm 5\%$, loss if HiPO is $> 5\%$ slower. `23_Softmax_base` is excluded *before* ranking and aggregation.