HARDWARE-IN-THE-LOOP RL FOR OPENCL OPTIMIZATION ON MOBILE GPUS

Anonymous authors
Paper under double-blind review

ABSTRACT

As large language models (LLMs) make rapid strides in code generation, excitement is growing about their potential to optimize system kernels, but the field's datasets and evaluation infrastructure are primarily anchored in the NVIDIA ecosystem. OpenCL offers a common programming interface across vendors whose portable implementations can benefit from hardware-specific optimizations. However, learning these optimizations requires executable, correctness-verifiable workloads and reliable feedback from the target device. We investigate whether LLMs can use live execution feedback during RL training to optimize OpenCL kernels for resource-constrained mobile GPUs. We build and open-source two datasets: **OpenCLKernelBench**, which adapts KernelBench Level 1 with OpenCL reference implementations, and **OpenCLOC**, 620 curated kernels mined from GitHub. Both pair kernel implementations with test cases and executable host code. We also present a framework for hardware-in-the-loop reinforcement learning with verified rewards (HIL-RLVR) that includes a custom training environment, protocols for quantifying and reducing device noise, and novel reward hacking interventions. Finally, we train models to optimize kernels for Adreno GPUs on Snapdragon mobile processors. Relative to their shared Qwen2.5-Coder-32B-Instruct base model, $\mathrm{fast}_{1.5}@8$ increases from 9% to 26% on OpenCLOC and from effectively 0% to 18% on OpenCLKernelBench, achieving performance competitive with GPT-OSS-120B on both datasets. We also train an OpenCL policy on an NVIDIA Jetson AGX Thor and find that the learned optimizations are hardware-specific; policies remain correct when evaluated on the other device type, but their speedups do not transfer. To our knowledge, this is the first application of LLM-based RLVR to OpenCL kernel optimization. Our datasets and framework make resource-constrained deployment hardware a practical source of feedback for learning kernel optimizations.

1 INTRODUCTION

Modern AI relies on fast kernels, but producing them remains labor-intensive and highly challenging even for human experts. Recent work has explored the promise of LLM-automated kernel optimization through execution-guided agents or finetuned models. Reinforcement learning with verified rewards (RLVR) is particularly well suited to leveraging direct execution feedback: proposed implementations can be checked for correctness and performance on the target hardware, providing a high-fidelity learning environment [15; 2; 17]. Much of the field's data and evaluation infrastructure has developed around KernelBench [15] and related benchmarks, with strong ties to PyTorch workloads and CUDA or Triton implementations in the NVIDIA ecosystem [15; 9]. These benchmarks have provided a valuable testbed for LLM-driven kernel optimization, but do not easily support studying other execution environments. Coverage is expanding, yet remains fragmented across programming languages, compiler toolchains, and runtime APIs [20; 11]. The resulting integration burden makes it hard to investigate hardware-specific optimization beyond established platforms, including the resource-constrained devices on which increasingly capable AI systems must run.

OpenCL offers a common starting point: its programming model supports portable kernel implementations without requiring a different source language for each hardware vendor. Portability does not, however, determine which implementation choices will make a kernel fast on a particular device. This leaves an exciting opportunity for learning from the target hardware itself. Realizing that

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.

opportunity requires overcoming several practical challenges. First is the lack of existing datasets, which requires not only collecting OpenCL kernels but also equipping them with correctness checks and executable host code so they can serve as optimization tasks. Building these workloads is essential both to learning meaningful reward signals and comparing methods under consistent evaluation conditions. Secondly, hardware-in-the-loop training must coordinate model generation, compilation, verification, and policy updates across machines with very different resources and execution latencies. It must also distinguish genuine improvements from spurious measurement errors. Clock behavior, thermal state, caching, and measurement overhead can all change a kernel's apparent performance; in an RL loop, those effects can reward code changes that do not actually improve execution. Mobile devices add tight resource budgets and limited control over operating conditions, making reliable, high-throughput evaluation especially demanding. Meanwhile, weaknesses in correctness checks can reward implementations that bypass the intended computation [21; 12].

We address each of these barriers with an end-to-end approach to hardware-in-the-loop reinforcement learning with verified rewards (HIL-RLVR), demonstrating that models can learn effective OpenCL kernel optimizations from live feedback on resource-constrained mobile GPUs. Our contributions include the construction of two open-source datasets of executable workloads, a highly efficient training infrastructure, and the measurement methodology required to provide a foundation for further hardware-specific kernel optimization experiments. We demonstrate meaningful optimization gains from device-guided training. Using single-turn Group Relative Policy Optimization (GRPO), we train Qwen2.5-Coder-32B-Instruct models to optimize kernels for Adreno GPUs on Snapdragon mobile processors. Training increases $\mathrm{fast}_{1.5}@8$ from 9% to 26% on OpenCLOC and from effectively 0% to 18% on OpenCLKernelBench, achieving performance competitive with GPT-OSS-120B on both datasets. To test whether these gains are specific to the training hardware, we also train a policy with feedback from NVIDIA Jetson AGX Thor boards and evaluate each policy on both devices. Correctness transfers (e.g. correct@1 of 0.87 on both device types for the Thor-trained policy), but speedup performance does not transfer, falling from 0.15 to 0.00 for the Thor-trained policy on Adreno, and from 0.25 to 0.05 for the Adreno-trained policy on Thor. These results demonstrate that deployment hardware can serve as an effective teacher for kernel optimization, not merely a destination for generated code.

2 Related work

2.1 Datasets and benchmarks

NVIDIA ecosystem. KernelBench [15] is a widely used benchmark for LLM kernel optimization. Its first two levels contain 200 tasks comprised of a PyTorch operator or short operator composition for which the LLM must write an optimized CUDA implementation. It also supports Triton [18] and several other domain specific languages (DSLs) popular in deep learning. Dedicated Triton benchmarks include TritonBench [9], made up of 184 operators mined from GitHub and 166 tasks synthesized by composing PyTorch operators, and RealisticTritonBench [7], whose 31 tasks are derived from real pull requests. These benchmarks advance LLM-driven kernel optimization primarily within the NVIDIA ecosystem. Extending to other hardware requires kernel implementations and execution environments that are not bound to the CUDA software stack.

Other datacenter hardware targets. MultiKernelBench [20] spans 285 deep learning kernel tasks targeting Huawei Ascend NPUs and Google TPUs. AMDKernelVault [11] provides a large HIP/Triton training corpus validated on AMD GPUs, combining kernels generated from PyTorch references with production-grounded supervision from ROCm libraries, and KernelBench has also extended support to certain AMD architectures. Other resources include NPUKernelBench [3], EvoKernel [24], and CANN Bench [5] for Ascend, JAXBench [19] for TPUs, and NKIBench [22] for AWS Trainium. These individually broaden hardware coverage, but supporting additional platforms continues to necessitate substantial platform-specific implementation and integration work, since even unified benchmarks still require separate kernel implementations and execution paths for each target platform.

OpenCL standard. OpenCL offers a cross-vendor programming interface spanning server, consumer, and embedded devices. It is also supported by NVIDIA. Its functional portability does not guarantee performance portability, making it a natural setting for learning hardware-specific optimizations from execution feedback. LLMPerf [13] open-sources a large-scale OpenCL dataset for

predicting runtime from a kernel and its test case. Its reported task count includes repeated profiling under 200 launch configurations per kernel, rather than counting unique implementations. Although it supplies argument metadata, its test cases are not verified for non-deterministic outputs or for distinguishing the original kernel from a no-op, limiting their use for correctness verification.

2.2 LLM-BASED KERNEL OPTIMIZATION

RLVR with hardware-in-the-loop. The closest works to ours uses RL to train LLMs for GPU kernel optimization using execution on the target hardware as a source of verified rewards. Kevin [2] introduces multi-turn RL for CUDA kernel generation, allowing the model to iteratively refine its kernels using execution feedback, while CUDA-L1 [17] develops a contrastive RL approach for CUDA optimization, learning from relative performance among generated variants. AutoTriton [10], TritonRL [21], and Dr. Kernel [12] target Triton kernel generation using a combination of supervised fine-tuning and RL with execution-based rewards, though AutoTriton focuses only on correctness. AMDKernelVault [11] extends the paradigm to include HIP and Triton on AMD MI-series GPUs. Collectively, these studies establish execution-guided RL as a promising approach to kernel optimization, but primarily study CUDA/Triton on datacenter GPUs.

Edge and consumer hardware targets. MobileKernelBench [25] investigates LLM-based kernel optimization for mobile devices, but focuses on inference operator kernels evaluated through MNN's CPU backend rather than low-level GPU kernels. Metal-Sci [4] introduces ten scientific-computing tasks targeting Metal kernels on Apple Silicon GPUs and explores LLM evolutionary search.

3 OPENCL DATASET CREATION

We introduce two OpenCL datasets geared toward complementary optimization tasks. OpenCL Optimized Corpus (OpenCLOC) contains 620 kernels curated from GitHub and targets kernel optimization given a fixed host program. OpenCLKernelBench is our OpenCL translation of KernelBench [15] Level 1, supporting joint optimization of host and device code.

3.1 OPENCLOC

The bulk of OpenCLOC kernels are sourced from an aggressively filtered subset of the kernels in LLMPerf [13], which are themselves derived from GitHub scraping. We describe our initial round of filtering, de-duplication, and post-processing steps in Appendix C.1.

Unlike LLMPerf's runtime-prediction setting, our task requires tests that reliably establish semantic equivalence between the original kernel and an LLM rewrite while providing meaningful optimization workloads on the target hardware. LLMPerf also does not identify which output buffers to read back for correctness checks. We use LLMs to generate semantically meaningful test cases with suitable workload sizes, then validate them to reject workloads that the original kernel cannot execute, exhibit nondeterminism, or fail to distinguish non-equivalent rewrites. The model receives multiple attempts to produce a valid specification, tested with random buffer data; kernels without a valid specification are discarded. Appendix C.2 details test generation and validation.

Our process yields 575 kernels in total from LLMPerf, which we augment with 45 newly scraped kernels that also pass our criteria. We identify 29 kernels having a 10x or greater speedup under our base model and manually audit these to remove an additional 21 kernels from the dataset (20 of which come from LLMPerf), resulting in a final total of 620 kernels. We reserve 100 kernels for evaluation, sampled uniformly across five difficulty levels that are based in how frequently the base, untrained model is able to write a functionally equivalent kernel.

3.2 OPENCLKERNELBENCH

OpenCLKernelBench is our OpenCL adaptation of KernelBench L1, whose tasks are individual operators. KernelBench is built around PyTorch modules, with CUDA (or supported DSLs such as Triton) implementations that run inside of a PyTorch program. This requires the target device to run PyTorch with a GPU backend, which in practice is NVIDIA GPUs through CUDA. OpenCL is a vendor-neutral standard: each task runs on any supported OpenCL device, so the source can be

tuned per device rather than written for vendor-specific features. Each of our 100 tasks consists of one to three OpenCL C kernels paired with C++ host code. We start with the KernelBench PyTorch modules, and derive CUDA host code for each task with static analysis, using an LLM only to recover buffer sizes the analysis cannot trace. We then translate the kernels to OpenCL C and the host code to C++ against the standard Khronos OpenCL C++ bindings. This translation, which involves deterministic and manual steps but no LLM usage, is detailed in Appendix D.

We rigorously verify each translated task for correctness against the reference implementation, determinism under repeated executions, and expected runtime behavior. The host program is cross-compiled for aarch64 with the Android NDK and run on Snapdragon devices with Adreno GPUs. The kernel is compiled on the device by the vendor's OpenCL driver at run time, as it would be in deployment. The OpenCL program receives the same inputs used to generate the CUDA reference outputs and a translated task passes only if every CUDA reference case has a corresponding OpenCL output and all compared cases pass. Because out-of-bounds accesses did not fault on our Adreno devices, we also ran each task on an NVIDIA Jetson AGX Thor using PoCL with a CUDA backend, comparing each baseline against an identical copy of itself on the same inputs to expose memory faults and run-to-run nondeterminism. Full acceptance of all 100 tasks was reached iteratively. Implementations that used CUDA features without an OpenCL counterpart were replaced with portable implementations of the same operation, and failed tasks were repaired rather than excluded. Task-specific changes and verification details are described in Appendix D.

4 Framework for accurate performance evaluation

Stable training requires reward signals that reflect kernel performance rather than fluctuations in device behavior. In this section, we describe our efforts to quantify the noise inherent in measuring execution time on our target hardware and reduce its impact on the quality of reward signals and evaluation metrics, as well as how we mitigate various reward hacking channels.

4.1 Reducing hardware effects

During a training run, candidate programs are evaluated on a pool of Qualcomm Snapdragon phones of the same model (either OnePlus15 or OnePlus15R). GPU clock behavior and power constraints can affect kernel timing, motivating explicit control of clock settings during evaluation [16]. We inspect each phone's supported GPU frequencies and driver-reported thermal caps, then select the highest frequency available to every phone in the pool. For the reported experiments, all OnePlus 15R devices are pinned to 1225 MHz and all OnePlus 15 devices are pinned to 726 MHz. We disable GPU frequency scaling and verify the settings prior to each run.

We then conduct a thorough hardware profiling phase in order to characterize device noise and reduce its impact on the accuracy of timing measurements and their downstream metrics. On each of the N_{device} devices, and independently for every kernel in the full dataset of size N_{kernel}, we measure 100 sequential timings, $t_{1:100}$. We then calculate and plot several aggregate statistics using the $N_{\text{device}} * N_{\text{kernel}} \times 100$ matrix of timings, described in the following subsections.

4.1.1 Warmup and repeated timings

To reduce non-stationary behavior during device warmup, it is common to perform a certain number of warmup runs before timed measurements. However, the number of warmup runs n_{warm} to discard and power-warmup timing runs n_{timed} to collect and aggregate over is not standardized across the literature, even under the same dataset and hardware setup. We derive empirically-motivated values for n_{warm} and n_{timed} by first normalizing every row by its converged tail, $median(t_{80:100})$, such that the normalized column i contains every kernel-device pair's divergence at time t_i from its eventual steady state. Taking the median and 10-90 percentile bands (Figure 1, left), we see that values fluctuate dramatically before converging sharply after approximately $n_{\text{warm}} = 20$ timings.

Next, using our previously found n_{warm} value, we select increasingly wide timing slices that start from n_{warm} and end at $j \in [n_{\text{warm}} + 1, 100]$. For each slice, we examine how the standard error of the mean[1] for a given kernel-device pair decreases with the inclusion of additional timings, finding

[1] We analyze both the standard form and the Newey–West HAC variant to account for correlated timings.

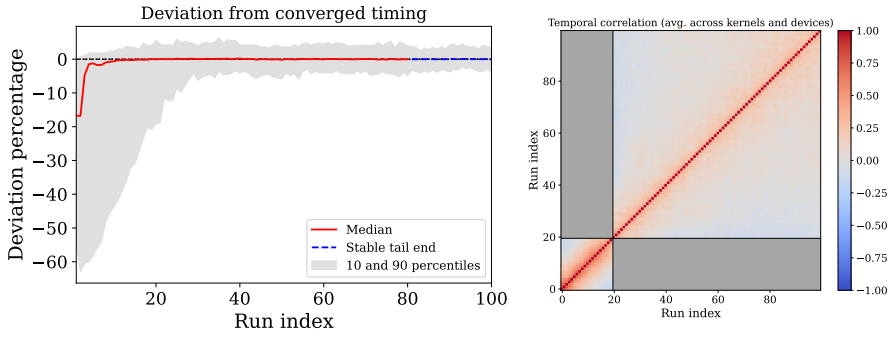


Figure 1: Convergence of (left) and temporal correlation between (right) repeated timing runs.

that it remains below 3% and relatively stable across timing runs after warmup. We select a conservative value of $n_{\text{timed}} = 30$, beyond which we are confident that further repeated timings provide diminishing returns on stability.

4.1.2 Interleaved timing strategy

The naive choice for benchmarking optimized kernels against their baselines is to repeatedly time each in isolation and then compare a statistic such as the mean over either runtime distribution. However, changes in on-device conditions such as temperature can lead to noise-dominated speedup measurements that make even identical implementations appear meaningfully different [1; 16]. Using our device profiling dataset, we plot the temporal correlation of timing order positions, where each pixel is the pairwise correlation between the normalized timings for that 2D timing position bin, averaged across all baseline kernels and devices (Figure 1), right). We see that sequential runs of the same kernel are indeed temporally correlated, particularly during the warmup period, after which correlation is mainly among neighboring consecutive measurements.

In order to reduce the confounding effect of varying device conditions when measuring each candidate's speedup, we interleave each of its n repeated timings with a new timing of the baseline kernel, $\{(B_1, O_1), ..., (B_n, O_n)\}$, such that temporal dependence largely cancels out in each ratio. However, always ordering one or the other (baseline or optimized) first can still create a systematic bias, so we furthermore randomize the position of the baseline in each pair.

4.2 Quantifying speedup

A robust and interpretable measure of speedup is necessary both for a strong reward signal and for meaningful evaluation and comparison of methods. Typical estimators for speedup involve the ratio of the baseline kernel time and optimized kernel time. Given that each kernel is profiled n times, there are multiple ways of aggregating this ratio over $t = \{1, ..., n\}$ timings for a given baseline kernel k and optimization candidate c, but the most common choice is the geometric mean of ratios,

$$\hat{S}_{kc}^{\text{gmor}} = \exp\left(\frac{1}{n}\sum_{t=1}^{n}\ln(B_{kct}/O_{kct})\right) = \left(\prod_{t=1}^{n}\frac{B_{kct}}{O_{kct}}\right)^{1/n}, \tag{1}$$

which has the attractive feature of log-scale symmetry (2× speedup and 0.5× slowdown have equal impact). Speedup must also be aggregated across all test cases for a given kernel. Our pipeline aggregates performance over five test cases per kernel and varies input coverage.

Downstream, evaluation reports typically include some subset of average, median, or maximum speedup across the dataset, as well as average fast$_p$@k for typical values of p and k. The fast$_p$@k metric, originally defined in KernelBench, extends the popular pass@k estimator for code correctness to also gate success on a minimum speedup factor p:

$$\text{fast}_p@k = \begin{cases} 1 & \text{at least one of } k \text{ is correct and has speedup} > p \\ 0 & \text{else} \end{cases} \tag{2}$$

Unlike correctness, speedup is a continuous metric, made binary via thresholding. Since fast_p@k is generally calculated with respect to a noisy point estimate of a completion's speedup—already collapsed into a single aggregate over timings and test cases—candidates whose true speedup lies near the threshold may be incorrectly scattered over or under it by a few noisy outliers. A more robust approach is to retain the individual timings and propagate their associated uncertainty. In our analysis, we calculate a bootstrap-smoothed version of fast_p@k, described in detail in Appendix E.1.

4.3 Robust reward signals

4.3.1 Reward hacking and candidate verification

Prior work identifies reward hacking as a recurring failure in execution-guided kernel optimization [2; 17; 21; 12; 23]. In particular, KernelBench-based studies have observed models bypassing kernel generation by wrapping PyTorch API calls or defining custom kernels that are never used. We do not observe this reward hacking mode. OpenCLKernelBench programs are standalone C++ and OpenCL, and every correct candidate in our logs launches all its kernels. In both OpenCLOC and OpenCLKernelBench tasks, however, we observe exact copying of baseline kernel implementations during training. We implement a copy detector that compares normalized abstract syntax trees (ASTs) of the baseline and candidate kernel code, and assign zero reward to any copied kernels. This prevents copies from erroneously being attributed a positive advantage due to pure noise in the speedup ratio. In OpenCLKernelBench, where host code can also be modified, we permit a candidate to retain either the host code or kernel code unchanged, but not both.

In addition, randomly generated test puts can be invalid even when the overall test spec is valid, such as inputs that cause under or overflow, and we observed examples where candidates passed correctness by outputting NaNs that matched the NaNs in the baseline, while also registering significant speedup due to bypassing the intended computation. Related work similarly finds that narrow test distributions allow generated programs to bypass required computation while passing standard correctness checks [23]. To mitigate this, we enforce operator-specific input ranges for test generation.

4.3.2 Reward shape

The model receives a positive reward when it produces a kernel whose output agrees with the baseline on all test cases and runtime is at least as fast without being a superficial copy:

$$R_{kc} = \begin{cases} \log(\langle \hat{S}_{kc} \rangle_{\text{tests}}) & \text{candidate is correct and } \langle \hat{S}_{kc} \rangle_{\text{tests}} \geq 1 \\ 0 & \text{candidate is incorrect or is a copy of the baseline} \end{cases} \tag{3}$$

We clip the speedup ratio from below at one so that the model does not collect positive advantage by producing correct but slower kernels. Our early experiments showed empirically that rewarding correct but slower kernels led the model to prefer safer yet less effective optimizations. Other experiments showed that a linear speedup reward encouraged the model to favor large measured speedups at the expense of reliability. This is particularly undesirable in our setting, where transient device effects can occasionally produce spuriously large speedups that are difficult to reproduce.

5 Efficient hardware-in-the-loop training

5.1 Interleaved training and verification

Standard GRPO training loops run the following phases sequentially: generate a batch of completions, score all of them, take one or more optimizer steps on the scored batch, then generate a new batch of completions from the updated model. Computing verified rewards based on hardware is slow and latency is highly variable across candidates, causing the training GPUs to sit idle while an entire generation batch is verified. In contrast, verification of candidates in our custom training environment is concurrent with gradient accumulation, so that GPUs wait for at most one microbatch of

verified candidates before performing backpropagation. A per-rank coordinator implements this as a multi-threaded producer–consumer pipeline. Each training rank manages a subset of verification devices and submits its candidate batches to a shared queue. Candidates are submitted to the shared device queue in column-stride order, so the completions within each GRPO group are spread across all of a rank's devices. Figure 5.1 shows a representative snapshot from our custom trainer's visualization callback, in which a single GPU with a pool of two verification devices performs gradient accumulation concurrently with kernel verification over two optimizer steps.

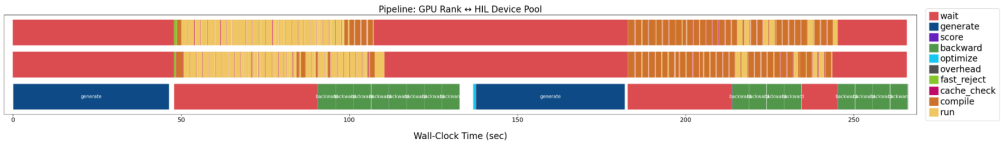


Figure 2: Training pipeline visualization over two optimizer steps. Hardware-based kernel verification is concurrent with gradient accumulation.

5.2 On-device efficiency

The training server communicates with the rooted phones over SSH through a private network. C++ host code is cross-compiled on the server with Android NDK while OpenCL kernels are compiled on the phone by the vendor driver at runtime. Cross-compilation on the training server allows compilation and execution to overlap for a single device.

Additionally, verification is staged, starting with fast-rejects of invalid candidates based on structural and syntax checks on the training server. These detect malformed or missing code, unsupported device features, and changes to the required host-function interface. Only candidates that pass these checks compiled and transferred to the device for correctness checking and timing, and we only time only kernels whose outputs are correct on all test cases to avoid unnecessary execution time on incorrect candidates.

6 Results on Snapdragon mobile devices

We train a Qwen-2.5-Coder-32B-Instruct [8] base model for each of our two tasks: kernel-only optimization on the OpenCLOC dataset and host-kernel optimization on the OpenCLKernelBench dataset, both using Snapdragon mobile devices as the target hardware. Details on the training setup, including hyperparameters, are given in Appendix A. Models are evaluated on the held-out test split of their respective task datasets. In order to rigorously evaluate our trained models, we aggregate results across five distinct train-evaluate protocols that span two differently seeded training runs, two different seeds used for generation during evaluation, and an evaluation run performed on a different subset of Snapdragon devices than the subset used as the training pool. We report the mean and 95% confidence interval values of each metric over these five protocols.

Performance is compared against the base Qwen-32B model as well as two frontier open-weight models, GPT-OSS-120B [14] and GLM-5.3-744B [6]. The reported metrics for baselines are aggregated across two evaluation runs that use different random seeds for generation and different device pool candidates for timing. The full results are given for OpenCLOC in Section 6.1 and OpenCLKernelBench in Section 6.2, but across both, our hardware-in-the-loop trained model significantly outperforms the base model and achieves competitive or superior performance to GPT-OSS-120B, though GLM-5.3-744B dominates the rest by a substantial margin.

6.1 OpenCLOC

Table 1 compares the base, trained, and frontier models on OpenCLOC. Training improves correctness and optimization coverage, achieving performance comparable to GPT-OSS-120B at roughly a quarter of its size. Stronger results from larger models suggest further headroom for RLVR-based optimization. Figure 6.1 shows progress throughout training. Discovery of optimizations for new kernels slows after the first epoch but continues with repeated exposure (left). The probability of sampling a fast kernel increases across all evaluated sampling budgets (center and right).

Table 1: Results on held-out evaluation set of 100 kernels from OpenCLOC. We provide mean values and 95% confidence intervals for all metrics.

Model	correct@1 ↑	mean S_{gmor} ↑	fast$_{1.1}$@8 ↑	fast$_{1.5}$@8 ↑
Qwen-Coder-32B-Instruct (base)	$0.27^{0.28}_{0.27}$	$1.19^{1.26}_{1.12}$	$0.18^{0.18}_{0.18}$	$0.09^{0.11}_{0.07}$
Trained model	$0.61^{0.65}_{0.57}$	$1.30^{1.37}_{1.24}$	$0.44^{0.48}_{0.41}$	$0.26^{0.29}_{0.24}$
GPT-OSS-120B	$0.48^{0.50}_{0.47}$	$1.63^{1.64}_{1.63}$	$0.43^{0.44}_{0.41}$	$0.29^{0.31}_{0.27}$
GLM-5.3-744B	$0.61^{0.62}_{0.61}$	$1.91^{1.92}_{1.90}$	$0.55^{0.57}_{0.54}$	$0.42^{0.43}_{0.42}$

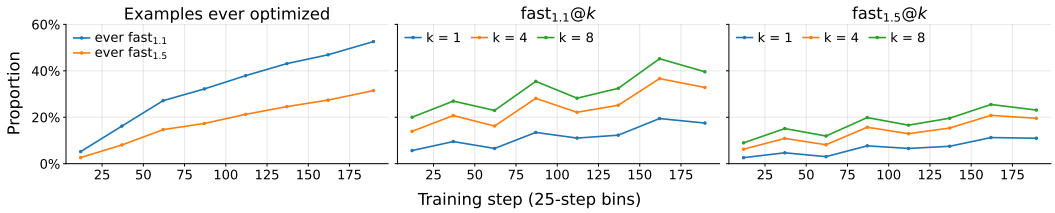


Figure 3: Speedup metrics during training on OpenCLOC. Samples where a speedup was ever obtained (left), average fast$_{1.1}$@k (center), and fast$_{1.5}$@k (right).

6.2 OpenCLKernelBench

Table 2 shows that training substantially improves performance on OpenCLKernelBench: correct@1 increases from 0.27 to 0.61, while fast$_{1.1}$@8 and fast$_{1.5}$@8 increase from 0.18 to 0.44 and from 0.09 to 0.26, respectively. The trained model exceeds GPT-OSS-120B in correctness and achieves comparable optimization coverage at the 1.1× threshold, although its mean speedup remains lower. Larger frontier models achieve higher mean speedups and broader coverage at the 1.5× threshold.

Figure 6.2 shows continued discovery of optimizations throughout training, with approximately 52% of training examples eventually exceeding 1.1× speedup and 31% exceeding 1.5×. Both fast$_{1.1}$@k and fast$_{1.5}$@k improve overall for all sampling budgets shown.

Table 2: Results on held-out evaluation set of 100 kernels from OpenCLKernelBench. We provide mean values and 95% confidence intervals for all metrics.

Model	correct@1 ↑	mean S_{gmor} ↑	fast$_{1.1}$@8 ↑	fast$_{1.5}$@8 ↑
Qwen-Coder-32B-Instruct (base)	$0.294\ ^{0.373}_{0.214}$	$1.055\ ^{1.088}_{1.022}$	$0.050\ ^{0.050}_{0.050}$	$0.000\ ^{0.000}_{0.000}$
Trained model	$0.800\ ^{1.000}_{0.324}$	$1.188\ ^{1.568}_{0.809}$	$0.400\ ^{0.400}_{0.400}$	$0.175\ ^{0.493}_{0.000}$
GPT-OSS-120B	$0.231\ ^{0.390}_{0.072}$	$1.010\ ^{1.269}_{0.751}$	$0.175\ ^{0.493}_{0.000}$	$0.100\ ^{0.100}_{0.100}$
GLM-5.3-744B	$0.700\ ^{0.779}_{0.621}$	$1.617\ ^{1.837}_{1.398}$	$0.825\ ^{1.000}_{0.507}$	$0.625\ ^{0.943}_{0.307}$

7 Transfer across hardware targets

To probe whether hardware-in-the-loop training is learning device-specific optimizations, we additionally train a second policy with feedback from a different GPU architecture, NVIDIA Jetson AGX Thor (setup details given in Appendix B.1). We use four Jetson Thor boards, one assigned to each training rank, and the same model initialization and training settings. Under the same OpenCLKernelBench training and held-out tasks, we evaluate the model trained to optimize for Snapdragon on the Thor devices and vice versa, with results shown in Table 3.

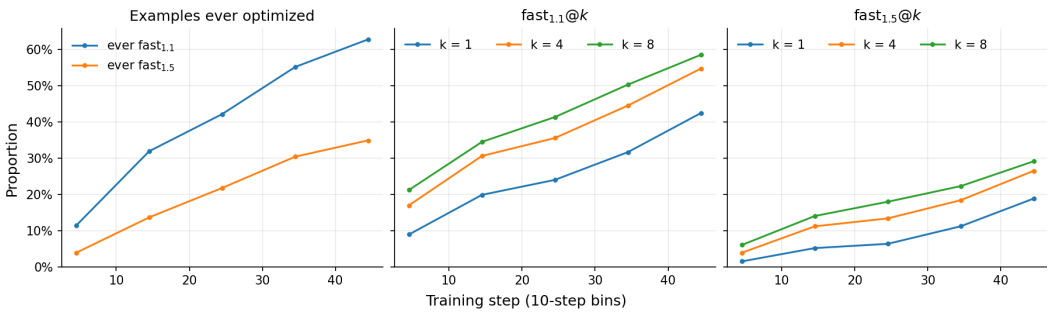


Figure 4: Speedup metrics during training on OpenCLKernelBench. Samples where a speedup was ever obtained (left), average $\text{fast}_{1.1}@k$ (center), and $\text{fast}_{1.5}@k$ (right).

Table 3: Held-out results at checkpoint 50: 20 tasks, 8 samples per task.

Train	Eval	correct@1 ↑	mean S_{gmor} ↑	$\text{fast}_{1.1}@8$ ↑	$\text{fast}_{1.5}@8$ ↑
Thor	Thor	0.869	1.128	0.150	0.150
Thor	Adreno	0.869	0.997	0.000	0.000
Adreno	Thor	0.806	1.101	0.250	0.050
Adreno	Adreno	0.838	1.217	0.400	0.250

While both policies learn correctness, they each dramatically underperform in producing optimized kernels for the non-target hardware compared to the target hardware they were trained on. We also observe that the transferability of policies is not symmetric, with models trained on Adreno performing better on Thor than vice versa.

8 Conclusions and Future Work

Our results show that reinforcement learning with on-device verified rewards can teach LLMs to optimize OpenCL kernels for mobile device GPUs. To our knowledge, ours is the first work to apply LLM-based RLVR to OpenCL kernel optimization and demonstrate its effectiveness on resource-constrained target hardware. Our framework for efficient training and robust rewards using hardware-in-the-loop represent a foundation for hardware-specific optimization of portable kernels that we hope to build on in subsequent work. Our initial results on a 32B parameter model are already competitive with some large frontier models, while the remaining gap to others suggests that further performance gains are available. In addition to scaling up our training experiments, we plan to extend cross-platform evaluation to explore which learned strategies generalize across architectures and runtimes.

AI use statement

In this work, we used generative AI tools to support qualitative and thematic data analysis. We did not use generative AI to design or provide feedback on research methodology or experiments, implement methods, assist with translation, clean or reformat datasets, or interpret results; other activities requiring disclosure under the ICLR policy were not applicable to this work. We additionally used generative AI tools to edit software code, draft portions of the paper, and improve the readability of the manuscript. All AI-assisted work was reviewed by the authors, including verification of AI-generated claims and testing of AI-generated code. The authors take responsibility for the final content of this work, including any text, claims, or artifacts produced with the aid of generative AI.

Reproducibility statement

Our datasets will be open-sourced after the review period (to preserve anonymity), and all details needed to reproduce our datasets are described in Section 3 and Appendices D and C. Our training and testing methodologies are described in full detail in Sections 4 and 5.

References

[1] Ali Abedi and Tim Brecht. Conducting repeatable experiments in highly variable cloud computing environments. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, pp. 287–292, New York, NY, USA, 2017. Association for Computing Machinery. doi: 10.1145/3030207.3030229. URL https://doi.org/10.1145/3030207.3030229.

[2] 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*, volume 2026, pp. 83418–83452, 2026.

[3] Xinzi Cao, Jianyang Zhai, Pengfei Li, Zhiheng Hu, Cen Yan, Bingxu Mu, Guanghuan Fang, Bin She, Jiayu Li, Yihan Su, et al. Ascendkernelgen: A systematic study of llm-based kernel generation for neural processing units. *arXiv preprint arXiv:2601.07160*, 2026.

[4] Víctor Gallego. Metal-sci: A scientific compute benchmark for evolutionary llm kernel search on apple silicon. *arXiv preprint arXiv:2605.09708*, 2026.

[5] Xue-Jian Gao, Deng Pan, Yueming Su, Jiasheng Li, Bin Du, Fengming Zhu, Chengdi Ma, Junyi Fan, Qichen Liao, Chengqiu Hu, et al. Cann bench: Benchmarking agent generated kernels against real npu and algorithmic limits. *arXiv preprint arXiv:2607.20518*, 2026.

[6] GLM-5-Team et al. GLM-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026. URL https://arxiv.org/abs/2602.15763.

[7] Jinjun Huang, Zhongzhen Wen, Tongtong Xu, Meng Yan, Xin Xia, and Zhongxin Liu. Realistictritonbench: A benchmark for triton-kernel generation in real-world ai frameworks. *arXiv preprint arXiv:2608.12004*, 2026.

[8] Binyuan Hui, Jian Yang, Zeyu Cui, et al. Qwen2.5-Coder technical report. *arXiv preprint arXiv:2409.12186*, 2024. URL https://arxiv.org/abs/2409.12186.

[9] Jianling Li, Shangzhan Li, Zhenye Gao, Qi Shi, Yuxuan Li, Zefan Wang, Jiacheng Huang, WangHaojie WangHaojie, Jianrong Wang, Xu Han, et al. Tritonbench: Benchmarking large language model capabilities for generating triton operators. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 23053–23066, 2025.

[10] Shangzhan Li, Zefan Wang, Ye He, Yuxuan Li, Qi Shi, Jianling Li, Yonggang Hu, Wanxiang Che, Xu Han, Zhiyuan Liu, et al. Autotriton: Automatic triton programming with reinforcement learning in llms. *arXiv preprint arXiv:2507.05687*, 2025.

[11] Ji Liu, Saptarshi Majumder, Yiqing Huang, Wenwen Ouyang, Umang Pandey, Zeping Li, Chushi Chen, Zihao An, Puyuan Yang, Zekai Li, et al. Amdkernelvault: Large-scale datasets and agentic training for amd gpu kernel optimization. *arXiv preprint arXiv:2609.12471*, 2026.

[12] 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. *arXiv preprint arXiv:2602.05885*, 2026.

[13] Minh-Khoi Nguyen-Nhat, Hoang Duy Nguyen Do, Huyen Thao Le, and Thanh Tuan Dao. Llmperf: Gpu performance modeling meets large language models. In *2024 32nd International Conference on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 1–8, 2024. doi: 10.1109/MASCOTS64422.2024.10786558.

[14] OpenAI et al. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*, 2025. URL https://arxiv.org/abs/2508.10925.

[15] Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? *arXiv preprint arXiv:2502.10517*, 2025.

[16] Standard Kernel. "this kernel was faster yesterday"—in pursuit of high-fidelity gpu kernel benchmarking. https://standardkernel.com/blog/in-pursuit-of-high-fidelity-gpu-kernel-benchmarking/, February 2026. Accessed: 2026-09-25.

[17] Xiaofei Sun, Albert Wang, Jiwei Li, Chris Shum, et al. Cuda-l1: Improving cuda optimization via contrastive reinforcement learning. In *International Conference on Learning Representations*, volume 2026, pp. 123922–123957, 2026.

[18] Philippe Tillet, Hsiang-Tsung Kung, and David D. Cox. Triton: an intermediate language and compiler for tiled neural network computations. *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2019. URL https://api.semanticscholar.org/CorpusID:184488182.

[19] Arya Tschand, Charles Hong, Julian Walker, Nina Cai, Shangkun Wang, Suvinay Subramanian, Sundar Dev, Vijay Janapa Reddi, Amir Yazdanbakhsh, and Sethu Sankaran. Jaxbench: Benchmarking autonomous tpu kernel optimization. *arXiv preprint arXiv:2607.20466*, 2026.

[20] 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.

[21] 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.

[22] 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. *Proceedings of Machine Learning and Systems*, 8:541–568, 2026.

[23] Yunxiang Zhang, Ping Yu, Jianyu Wang, Julian Reed, Azalia Mirhoseini, Will Su, et al. Kernelbench-verified: Do llm-generated kernels actually beat pytorch? *arXiv preprint arXiv:2607.16241*, 2026.

[24] Yujie Zheng, Zhuo Li, Shengtao Zhang, Hanjing Wang, Junjie Sheng, Jiaqian Wang, Junchi Yan, Weinan Zhang, Ying Wen, Bo Tang, et al. Towards cold-start drafting and continual refining: A value-driven memory approach with application to npu kernel synthesis. *arXiv preprint arXiv:2603.10846*, 2026.

[25] Xingze Zou, Jing Wang, Yuhua Zheng, Xueyi Chen, Haolei Bai, Lingcheng Kong, Syed AR Abu-Bakar, Zhaode Wang, Chengfei Lv, Haoji Hu, et al. Mobilekernelbench: Can llms write efficient kernels for mobile devices? *arXiv preprint arXiv:2603.11935*, 2026.

A TRAINING DETAILS

Training runs are performed on a server with eight NVIDIA B300 GPUs. Each training run uses four to eight GPUs with collocated vLLM inference. Candidate programs are evaluated on eight to sixteen Snapdragon phones, divided evenly across training ranks. OpenCLOC experiments use OnePlus 15R phones while OpenCLKernelBench experiments use OnePlus 15 phones.

Table 4: Default shared training parameters

PARAMETER	SETTING
Group size G	16
PPO ϵ	0.2
PPO iterations	1
KL coefficient	0.005
Advantage normalization	Within groups
Microbatch size	8 completions per GPU
Gradient accumulation steps	4
Steps per generation	4
LoRA rank	64
LoRA α	128
# GPUs	4
Microbatch size	8 per GPU
Gradient accumulation	4
Optimizer	AdamW
Learning rate	1×10^{-4}
Warmup steps	20
AdamW β_1	0.9
AdamW β_2	0.95
AdamW ϵ	1×10^{-8}
Weight decay	0.01
Gradient norm clip	1.0
Maximum completion length	4,096 tokens
Correctness test cases	5 per candidate
Training steps	205 (OpenCLOC), 50 (OpenCLKernelBench)

B HARDWARE DETAILS

B.1 JETSON THOR

Each board runs OpenCL through PoCL 7.2's CUDA backend, patched to support the half-precision vector stores and subgroup shuffle operations required by the kernels. We disable CPU devices to prevent silent CPU fallback and pin the GPU frequency at 1575 MHz.

C OPENCLOC CONSTRUCTION

C.1 QUALITY FILTERING AND POST-PROCESSING

The expanded version of LLMPerf contains 600k rows representing 4.2k unique kernels. We perform several post-processing steps:

- Strip helper functions that are not reachable from the kernel
- Split device code containing multiple kernels that do not call each other into separate examples
- De-duplicate based on exact match after stripping comments and normalizing variable names
- Filter by character length, keeping device-side code between 500 and 20k characters

- Filter to kernels whose arguments types are supported in our test case generation:
  - Char types: `char` and `unsigned char`
  - Integral types: `short`, `unsigned short`, `int`, `unsigned int`, `long`, and `unsigned long`
  - Float types: `float`
  - Vector type variants of the above of width 2, 3, 4, 8, and 16
- Remove a hand-curated list of kernel name patterns:
  - Remove kernels with names containing `test_`
  - Remove kernels whose names end with `kernel`, `kernel_<digits>`, `foo`, `bar`, `baz`, `qux`, `dummy`, `mykernel`, or `my_kernel`.
  - Remove kernels ending with `_withDD`, `_withoutDD`, `_withDD_<digits>`, or `_withoutDD_<digits>`
- Remove kernels with too many repeated lines: We classify a kernel as likely runtime-generated when it exceeds 5 repeated integer-normalized lines
- Keep only kernels for which the test case spec generation process desribed in Appendix C.2 succeeds

C.2 Test case spec generation and validation

We use GLM 5.2 with temperature 0.6 and a maximum completion length of 65k to generate semantically reasonable test case specs. The model is given the kernel body and is then instructed to output global and local sizes, a size for each buffer, a concrete value for each scalar, and to mark which buffers contain the kernel's output.

Each test case spec is carefully validated. First, we check for test cases that exhibit non-determinisim (multiple trials on the same kernel and test case input should produce equivalent output within error tolerances) or vacuity (a no-op kernel must fail the tests). Second, we require that the average runtime of the kernel is between 1-10 ms, to reduce the chances of the difference between baseline and optimized kernel timings falling below the timing measurement noise floor. Lastly, we limit the total size of a test case to 64 MB. The model receives feedback based on any violated constraints and is give 5 tries to find a passing test case spec. The initial instruction prompt and feedback prompts are in Appendix C.3

C.3 Test Case Spec Generation Prompt

Listing 1: Initial test case spec generation prompt used for the OpenCLOC dataset.

```
System:
## TASK DESCRIPTION
You are an expert in GPU programming. Your task is to size a test case
    for an OpenCL kernel so that it makes a good optimization benchmark
    on a Qualcomm Adreno GPU.

You will be given an OpenCL kernel and the types of its arguments.
    Analyze the kernel's arguments and body to work out what it computes
    and how it indexes memory. Then produce, IN ORDER:
1. A list of the output buffers -- every buffer the kernel writes to.
2. The global and local NDRange sizes.
3. `global_size_expr` -- one C expression per NDRange dimension computing
     that dimension from scalar
   parameter names. If `local_size` is set, round up (ex. `"((n + 255) /
    256) * 256"`). If
   `local_size` is null, use the scalar directly (ex. `"n"`).
4. A value for every scalar argument.
5. The number of elements in every input buffer.
6. The number of elements in every output buffer.
7. The number of elements in every local buffer.

## ACCEPTANCE CONSTRAINTS
```

```
Your test case is measured on a real Adreno GPU and is only accepted when
     ALL FOUR hold. You get up to 5 attempts, and after each one you are
    told exactly which constraint failed and by how much.

1. **Runtime band.** The kernel's measured execution time must land in
   **[1 ms, 10 ms]**; aim for **4 ms**. Too fast and the measurement
   is dominated by launch overhead and an optimization is unmeasurable;
    too slow and the benchmark is uselessly expensive. Runtime is driven
    by the total work: NDRange size x per-work-item work.
2. **Transfer budget.** Every buffer is uploaded to a phone over the
    network, so the sum over all
   buffers of `elements x sizeof(element)` must stay under **64 MB**.
    Leave headroom -- target about 80% of that -- because the packed blob
     carries a little framing beyond the raw elements, and a test case
    that lands exactly on the cap comes back over it.
3. **Discriminating output.** A do-nothing kernel -- same signature,
    empty body -- is run against your test case and its output is
    compared to the real kernel's, by relative L2 with a 1e-3 tolerance.
    The do-nothing kernel MUST fail. It passes when the real kernel
    leaves its declared output buffers essentially untouched, which
    happens if: the kernel only writes under a condition your scalar
    values never satisfy; it writes back what it read; you declared an
    output buffer the kernel never writes; or `global_size` covers so
    small a fraction of a large output buffer that the untouched
    remainder swamps the L2 norm. Every declared output buffer must be
    substantially overwritten.

4. **A launch earned by the data.** `global_size` must be set by how much
     output the kernel computes,
   not turned up until the clock reads right. This is checked: the kernel
     is re-run over a much
   smaller `global_size`, and if that produces identical output
    appreciably faster, your test case is
   rejected -- it would reward a model for simply shrinking the launch.
    Size `global_size` to the
   number of output elements the kernel actually computes (or that
    divided by the elements each
   work-item handles), and reach the runtime band by growing the *buffers
    *.

Constraints 1 and 2 pull against each other. A kernel that streams memory
     once per work-item cannot reach the top of the band inside the
    transfer budget -- for those, size buffers near the budget and accept
     the low end of the band. A kernel with an inner loop can reach the
    band at a modest buffer size by raising the loop's trip count through
     its scalar arguments.

### FIXED SHAPES
Emit **fixed** sizes only. Do NOT emit a `"range"` field on any scalar
    and do NOT emit a `"size_expr"` field on any buffer. Every buffer's `
    size` is a literal element count. (`size` counts *vector* elements
    for a vector-typed argument: a `float4*` buffer of `size` 1024 is 16
    KB.)

### Value ranges and preferred values per type
- char: -128 to 127
- uchar: 0 to 255
- short: -32,768 to 32,767
- ushort: 0 to 65,535
- int: -2,147,483,648 to 2,147,483,647
- uint: 0 to 4,294,967,295
- long: -2,147,483,648 to 2,147,483,647, and ulong: 0 to 4,294,967,295 --
     **32-bit bounds on purpose**.
```

```
  The C types are 64-bit and the kernel receives 8 bytes, but the test-
    input generator validates these two against 32-bit range and refuses
    anything wider, so stay inside it.
- float: ~-3.4e38 to ~3.4e38
For integral scalars that do NOT define the problem size, prefer values
    in -16 to 16. For float scalars, prefer values in -1 to 1. Any
    integer used to index or bound a buffer MUST agree with that buffer's
     declared size -- a scalar larger than the buffer it bounds reads out
     of bounds and the run fails.

### Important task details
- Answer with valid JSON between `<inference_results></inference_results
    >` tags.
- `global_size` / `local_size` are lists: `[dim_0]`, `[dim_0, dim_1]`, or
     `[dim_0, dim_1, dim_2]`.
  Use the minimum working dimension the kernel needs.
- Use `null` for `local_size` unless the kernel needs a specific work-
    group size (it does if it uses
  local memory, barriers, or `reqd_work_group_size`). When set, every `
    global_size` dimension must be
  an exact multiple of the corresponding `local_size` dimension, and
    their product must not exceed
  1024 -- a violation makes the launch fail outright.
- `global_size_expr` is compiled as C, so `/` is integer division and is
    fine. Use only scalar
  parameter names, integer literals, `+ - * /` and parentheses  no casts,
     no ternaries, no calls.
- `buffer_sizes` must have one entry per buffer argument, INCLUDING local
     buffers.
- `scalar_values` must have one entry per non-buffer argument. Together
    the two lists must name every
  kernel argument exactly once.

### IF THE KERNEL CANNOT BE SIZED
Some kernels cannot meet the constraints at any size. Answer with
`<reject>one-line reason</reject>` instead of `<inference_results>` when,
     and only when:
- The work is driven by the *contents* of an input buffer rather than by
    its size or by a scalar
  (the harness fills input buffers with small random values you do not
    control), or
- The kernel is structurally fixed to a tiny size (ex. `
    reqd_work_group_size(16, 1, 1)` with a body
  that indexes only within that group), so no argument makes it bigger.
Do not reject just because the first attempt missed the band  you will be
     told how far off it was.

## EXAMPLE INPUT
### KERNEL SOURCE
```opencl
__kernel void vectorAddAndScale(
    __global const float* a,          // Input
    __global const float* b,          // Input
    __global       float* result,     // Output
                const int  n,          // Scalar
                const float scale      // Scalar
) {
    int gid = get_global_id(0);
    if (gid < n) {
        result[gid] = (a[gid] + b[gid]) * scale;
    }
}
```
ARGUMENT INFO
- a is a global buffer of type float
```

```
- b is a global buffer of type float
- result is a global buffer of type float
- n is a scalar of type int
- scale is a scalar of type float
## EXAMPLE OUTPUT
(Illustrative, assuming a 64 MB budget -- use the budget stated above for
     the real kernel.)
Three float buffers of n elements cost `12n` bytes, so 64 MB allows n up
    to about 5.6 million. Each
work-item does two loads, one store and one multiply, so this is purely
    bandwidth-bound and will land
at the low end of the band -- expected and accepted for a kernel with no
    inner loop. Rounding down to
a power of two:
<inference_results>
{
    "output_buffers": ["result"],
    "global_size": [4194304],
    "global_size_expr": ["n"],
    "local_size": null,
    "scalar_values": [
        {"name": "n", "value": 4194304},
        {"name": "scale", "value": 0.5}
    ],
    "buffer_sizes": [
        {"name": "a", "size": 4194304},
        {"name": "b", "size": 4194304},
        {"name": "result", "size": 4194304}
    ]
}
</inference_results>

User:
## KERNEL SOURCE
```opencl
kernel void global_prefixSum(global unsigned int* buffer, const int
    offset, const int size) {
  int localSize = get_local_size(0);
  int groupIdx = get_group_id(0);

  int sortedLocalBlocks = offset / localSize;

  int gidToUnsortedBlocks = groupIdx
      + (groupIdx / ((offset << 1) - sortedLocalBlocks) + 1)
      * sortedLocalBlocks;

  int globalIdx = (gidToUnsortedBlocks * localSize + get_local_id(0));
  if (((globalIdx + 1) % offset != 0) && (globalIdx < size))
    buffer[globalIdx] += buffer[globalIdx - (globalIdx % offset + 1)];
}
```
ARGUMENT INFO
Kernel to size: `global_prefixSum`

- buffer is a global buffer of type unsigned int
- offset is a scalar of type int
- size is a scalar of type int
```

Listing 2: Feedback the model sees when the output cannot be parsed.

```
User:
I could not find an `<inference_results>` block in your reply. Answer
    with the JSON between `<inference_results>` and `</inference_results
    >` tags, or with `<reject>reason</reject>`.
```

Listing 3: Feedback the model sees when the output cannot be used to construct a test case spec.

```
User:
The test case could not be built from your output: {build_error}
```

Listing 4: Feedback the model sees when spec arguments do not match those of the kernel.

```
User:
The argument names do not match the kernel signature: {argument_error}
`scalar_values` and `buffer_sizes` together must name every kernel
    argument exactly once, with buffers (including local buffers) in `
    buffer_sizes` and non-buffers in `scalar_values`.
```

Listing 5: Feedback the model sees when the test case data is too large.

```
User:
Your buffers total {buffer_mb} MB. The budget is 64 MB. Scale the buffer
    sizes (and the scalars and global_size that go with them) down to fit
    .
```

Listing 6: Feedback the model sees when the kernel fails to run.

```
User:
The kernel failed to run on the device with this test case. This is
    usually an out-of-bounds access (a scalar bound larger than the
    buffer it indexes, or a global_size past the end of a buffer) or an
    invalid work-group size. Device output:
```
{device_output}
```
```

Listing 7: Feedback the model sees when a vacuous test case is detected.

```
User:
A do-nothing kernel -- your kernel's signature with an empty body --
    produced output within
tolerance of the real kernel's (relative L2 = {relative_l2}), so this
    test case cannot tell a working kernel from one that does nothing.
    Either the kernel does not write the buffers you declared as outputs
    on these inputs, or the launch covers too small a fraction of them
    for the difference to register. Check which buffers are actually
    written, pick scalar values that make the writes happen, and make
    sure `global_size` covers every element of every output buffer.
```

Listing 8: Feedback the model sees when the kernel output is non-deterministic.

```
User:
The kernel does not reproduce its own output. I ran your test case with
    the SAME kernel on both sides of the comparison -- starting from byte
    -identical input buffers, so it must agree with itself -- and it did
    not (relative L2 = {relative_l2}). Two things cause this:
- **A data race**: unsynchronized writes to the same location, a missing
    barrier, or atomics whose accumulation order varies between runs. No
    test case can fix that, and the kernel is unusable as a benchmark --
    answer with `<reject>`.
- **An out-of-bounds access** that doesn't crash: the kernel reads memory
     that isn't part of the test case and gets different garbage each run
    . That IS fixable -- check that every scalar bound agrees with the
    buffer it indexes, that `global_size` doesn't run past the end of any
     output buffer, and that local buffers are at least as large as the
    work-group indexes into them. Work out which one this is from the
    kernel source, then either fix the sizes or reject.
```

Listing 9: Feedback the model sees when the median timing is greater than 10 ms.

```
User:
Measured on the device: **{measured_ms} ms**. The band is [1 ms, 10 ms],
    aiming for 4 ms, so this test case does about **{factor}x too much**
    work.
Runtime scales with total work = NDRange size x per-work-item work, so
    you need roughly
{factor}x LESS of it. If the kernel has an inner loop, raising the scalar
     that bounds it scales the work without growing any buffer; otherwise
     grow the buffers and global_size together (staying under 64 MB) --
    and if you are already at the budget and still too fast, say so by
    proposing the largest test case that fits.
```

D OpenCLKernelBench construction

D.0.1 Stored tasks

Each task stores the OpenCL kernel, C++ host code, target function name, and resolved buffer and scalar metadata. It also records the CUDA kernel and driver from which it was derived. The surrounding harness supplies OpenCL context, queue, and program setup. At export, we check that the target function is present, its parameter count matches the resolved buffers and scalars, and the kernel name can be identified.

Tasks contain two to six buffers and exactly one output buffer. Of the 306 buffers, 302 use fp16, three use 64-bit integers, and one uses a boolean type. The median kernel and host code lengths are 55 and 120 lines, respectively.

D.0.2 Host-code verification

Before translation, a CUDA program calls the generated host function and the original launcher with identical inputs over three random seeds. Outputs must match bitwise or have a maximum absolute difference below 10^{-3} for fp16 and 10^{-6} for fp32. Empty outputs are treated as failures. This check tests whether host-code derivation preserves the original launcher's behavior. The separate libtorch driver compares the CUDA implementation with its PyTorch reference.

D.0.3 Shapes and test inputs

Each task uses a fixed problem shape. Test cases vary the input data rather than independently sampling dimensions, since dimensions can be coupled by constraints such as convolution kernel sizes and output shapes.

Problem sizes are reduced by repeatedly halving the largest dimension parameter until no buffer exceeds 2^{21} elements. Dimensions that refer to the same quantity must remain consistent across allocations and kernel arguments. In several tasks, two scalar names referred to the same dimension, but only one was reduced. This caused buffer overruns or incorrect indexing. These aliases were corrected, and ties between equal dimensions were resolved deterministically.

Some tasks required additional shape adjustments. Standard matrix multiplication was reduced from $M = 1024$, $K = 4096$, and $N = 2048$ to $M = 256$, $K = 1024$, and $N = 512$. Lower-triangular matrix multiplication was reduced from $N = 4096$ to $N = 512$. A transposed convolution also received a smaller shape. The reduced axis of the product-reduction task was shortened from 256 to 16 to reduce fp16 underflow.

Symbolic size expressions must preserve operator precedence and C integer-division semantics. Errors in these expressions can make allocated buffer lengths disagree with kernel indexing even when the individual scalar values are valid. Size expressions and the resulting allocations were corrected together.

The default floating-point input range is $[-2, 2]$. A wider range can produce fp16 overflow, while long products can underflow even when their inputs are representable. Matching non-finite or zero outputs can then conceal incorrect implementations.

We therefore added domain-specific input overrides. BatchNorm and KL divergence use inputs in $[0.1, 1.0]$ to avoid invalid variances or non-positive values. Product reduction and cumulative product use inputs in $[0.99, 1.01]$ to reduce underflow. Subnormal injection is disabled for these overrides. The default test generator otherwise includes subnormal values with probability 0.1. Random input generation is seeded.

D.0.4 Output verification and baseline host code

CUDA-to-OpenCL verification compares outputs on the same inputs. Finite values are compared using relative L2 error with a threshold of 10^{-3}. NaN positions and infinity signs must match. A case with no finite values to compare is rejected. This rule can still leave matching non-finite values unchecked, which motivates the domain-specific input overrides.

The baseline host code allocates buffers with read-write access, uses blocking transfers, and synchronizes after launches. These choices leave host-side optimization opportunities in addition to kernel optimization.

E Metrics details

E.1 Bootstrap-smoothed fast$_p$@k

F Optimization Prompts

Listing 10: Prompt used for the OpenCLOC dataset.

```
System:
You are an expert at optimizing OpenCL kernels on the target hardware
  device. You will be given an OpenCL kernel. A benchmark harness you
  cannot see allocates the buffers, binds the kernel's arguments and
  launches it; your only job is to make the kernel itself faster.

## TARGET HARDWARE
Target device: QUALCOMM Adreno(TM) 829 (OpenCL 3.0)
- 8 compute units, max work group size 1024
- 32 KB local memory, 5.4 GB global memory
- No double (fp64) support
- Barrier flags: CLK_LOCAL_MEM_FENCE, CLK_GLOBAL_MEM_FENCE
  (no CLK_LOCAL_MEM_FENCE_ALL)

## OPTIMIZATION GOAL
Rewrite the kernel so it runs as fast as possible while producing
  numerically equivalent outputs. The only requirements are:
- You must write exactly one kernel.
- Your optimized kernel must have the exact same argument list as the
  original.
  Changing the argument list will mean you failed the task.
- Correctness is checked per output buffer using a maximum relative L2
  error
  of 1e-3:
  ||optimized - original||_2 / ||original||_2 <= 1e-3, computed over
  finite pairs.
  If the original buffer has zero norm, the finite outputs must match
  exactly.
  NaNs must match NaNs, and infinities must match in sign.

## REQUIRED RESPONSE FORMAT
Your response must contain at least one OpenCL code block like the
  following. You may output multiple code blocks, but we will always
  evaluate the last one you output.

'''opencl
// Put modified kernel code here!
```

````
```

User:
## KERNEL TO OPTIMIZE
```opencl
kernel void global_prefixSum(global unsigned int* buffer,
                             const int offset, const int size) {
  int localSize = get_local_size(0);
  int groupIdx = get_group_id(0);

  int sortedLocalBlocks = offset / localSize;

  int gidToUnsortedBlocks = groupIdx
      + (groupIdx / ((offset << 1) - sortedLocalBlocks) + 1)
      * sortedLocalBlocks;

  int globalIdx = (gidToUnsortedBlocks * localSize + get_local_id(0));
  if (((globalIdx + 1) % offset != 0) && (globalIdx < size))
    buffer[globalIdx] += buffer[globalIdx - (globalIdx % offset + 1)];
}
```
````

Listing 11: Prompt used for the OpenCLKernelBench dataset.

```
System:
You are an expert at optimizing OpenCL kernels and host code on the
  target hardware device.
You will be given two things:
1. OpenCL kernel code
2. Host code that allocates buffers, transfers data, and launches the
  kernel(s) on the device.

The host code contains a target function `hardsigmoid_target` that
  accepts host-side pointers and scalar parameters. This function
  manages device buffer allocation, host-to-device transfers,
kernel execution, and device-to-host readback. Helpers `_cl_init(path)
  `, `_cl_get_kernel(name)`, and globals `_cl_context` / `_cl_queue`
  are provided by the surrounding program.

## TARGET HARDWARE
Target device: QUALCOMM Adreno(TM) 829 (OpenCL 3.0)
- 8 compute units, max work group size 1024
- 32 KB local memory, 5.4 GB global memory
- No double (fp64) support
- Use CLK_LOCAL_MEM_FENCE and/or CLK_GLOBAL_MEM_FENCE as barrier flags
  (no CLK_LOCAL_MEM_FENCE_ALL)
## OPTIMIZATION GOAL
Modify the kernel code and host code so that `hardsigmoid_target` runs
  as fast as possible end to end (including data transfers). You may:
- Restructure the kernel (tiling, vectorization, memory access patterns
  , work-group optimizations)
- Modify kernel arguments (ex. introduce `__local` memory buffers)
- Change NDRange dimensions (global and local work sizes)
- Restructure buffer allocation and data transfer patterns
- Modify the launch function's internal logic

Requirements:
- The `hardsigmoid_target` function signature must remain exactly the
  same (same parameter names, types, and order)
- The target function must produce the same output as the original for
  the same input, within a relative L2 error of 1e-3
- The kernel name must remain `hardsigmoid_fp16_kernel`

## OPENCL API NOTES
```
```

````
- Use `CLK_LOCAL_MEM_FENCE` and/or `CLK_GLOBAL_MEM_FENCE` as barrier
  flags. There is no `CLK_LOCAL_MEM_FENCE_ALL`.
- Size `__local` kernel parameters with `_cl_get_kernel(name).setArg(
  idx, cl::Local(size_in_bytes))`.
- Declare `__local` buffers as kernel parameters or at kernel-function
  scope, never at file/program scope.
- Do not declare `__local` memory that you write to as `const`.
- Host code is compiled as C++, so OpenCL C types such as `uchar` or `
  float4` are not available in it.

## REQUIRED RESPONSE FORMAT
Output ONLY the two code blocks below. Do not include explanations,
  strategy descriptions, or commentary.
```opencl
// Put modified kernel code here!
```
```cpp
// Put modified host code here!
```

User:
## KERNEL CODE
```opencl
#pragma OPENCL EXTENSION cl_khr_fp16 : enable
#pragma OPENCL EXTENSION cl_khr_subgroups : enable
#pragma OPENCL SELECT_ROUNDING_MODE rte

__kernel void hardsigmoid_fp16_kernel(const __global half* restrict
  input, __global half* restrict output, long numel) {

    long idx = get_global_id(0);
    if (idx >= numel) return;



    float x = convert_float(input[idx]);
    float y = x * 0.2f + 0.5f;
    y = fmin(fmax(y, 0.0f), 1.0f);

    output[idx] = convert_half(y);
}

```
## HOST CODE
```cpp
#include <cstdio>
#include <unistd.h>
#include <numeric>
#include <algorithm>
#include <thread>
#include <chrono>

#include <cstdint>

void launch_gpu_implementation(
    cl::Buffer output,
    cl::Buffer input,
    int64_t batch_size,
    int64_t dim
) {
    _cl_init("./028_HardSigmoid.cl");

    int64_t numel = batch_size * dim;

    int block_size = 256;
````
````

```
    int grid_size = static_cast<int>((numel + block_size - 1) /
  block_size);


    {
        _cl_get_kernel("hardsigmoid_fp16_kernel").setArg(0, input);
        _cl_get_kernel("hardsigmoid_fp16_kernel").setArg(1, output);
        _cl_get_kernel("hardsigmoid_fp16_kernel").setArg(2, numel);

        cl::NDRange globalDims(grid_size);
        cl::NDRange localDims(block_size);
        cl::NDRange totalGlobalDims(
            globalDims[0] * localDims[0],
            globalDims[1] * localDims[1],
            globalDims[2] * localDims[2]);


        _cl_queue.enqueueNDRangeKernel(_cl_get_kernel("
  hardsigmoid_fp16_kernel"), cl::NullRange, cl::NDRange(totalGlobalDims
  ), cl::NDRange(localDims));
        _cl_queue.finish();
    }

        _cl_queue.finish();

}

void hardsigmoid_target(
    const half* host_input,
    half* host_output,
    int batch_size,
    int dim
) {
    half* host_output,
    int batch_size,
    int dim
) {
    half* host_output,
    _cl_init("./028_HardSigmoid.cl");

    const int input_count = batch_size * dim;

        cl::Buffer dev_input(_cl_context, CL_MEM_READ_WRITE,
  input_count * sizeof(half));

    const int output_count = batch_size * dim;

        cl::Buffer dev_output(_cl_context, CL_MEM_READ_WRITE,
  output_count * sizeof(half));

        _cl_queue.enqueueWriteBuffer(dev_input, CL_TRUE, 0, input_count
   * sizeof(half), host_input);

    launch_gpu_implementation(
        dev_output,
        dev_input,
        static_cast<int64_t>(batch_size),
        static_cast<int64_t>(dim)
    );

        _cl_queue.enqueueReadBuffer(dev_output, CL_TRUE, 0,
  output_count * sizeof(half), host_output);
    _cl_queue.finish();

}
```

Table 5: Best correct speedup (×) of 8 samples per held-out task at checkpoint 50 with the corrected harness, for tasks where any policy and device pool exceeds 1.2×. Bold marks speedups above 1.2×.

	Thor-trained policy		Snapdragon-trained policy	
Task	Thor pool	Snapdragon pool	Thor pool	Snapdragon pool
001 Square matrix multiplication	**1.62**	1.05	1.00	0.99
003 Batched matrix multiplication	**1.61**	0.97	1.04	1.00
018 Matmul, both inputs transposed	**3.78**	0.96	1.00	1.07
019 ReLU	1.07	1.01	1.10	**1.47**
020 LeakyReLU	1.04	1.00	1.05	**1.45**
029 Softplus	1.04	1.01	1.11	**1.45**
064 Transposed convolution 1D	1.07	1.02	1.19	**1.51**
065 Transposed convolution 2D	1.00	1.00	**4.01**	**4.03**
073 Transposed convolution 3D	1.04	1.00	**1.47**	**1.56**
093 Masked cumulative sum	1.02	1.01	1.09	**1.66**
099 Triplet margin loss	1.04	1.00	1.03	**1.51**

```
```

G Cross Architecture Experiment Details

Here, we describe a reward-hacking behavior that surfaced in our first Thor run, the harness changes we made in response, and per-task speedups for each training and evaluation device (Table 5). The Thor policy reported in the main text was retrained after these changes.

In our initial Thor run, the policy learned to remove the host code's synchronization, deleting the command queue `finish` calls and making buffer transfers non-blocking, which exposed a gap in our harness. On Thor, PoCL had already completed the copy when outputs were checked, so these programs passed correctness and the skipped waits counted as speedup, while on the Snapdragon, the same programs failed because the output had not yet been written. All results in this section use a harness that waits for all queued work before stopping the timer and checking outputs, and rejects host code that creates its own command queues or threads. Because host code controls synchronization, data movement, and command submission, the policy has many ways to change what is measured without changing the computation. This makes enforcing the program's full host contract, without the model learning to exploit it, especially difficult.

We see in 5 that each policy's largest speedups fall on different tasks and mostly do not carry over to the other device. The Thor-trained policy gains on matrix multiplications (up to 3.78×) do not appear on Snapdragon, and vice versa. The exceptions are two transposed convolutions (065 and 073), where the Snapdragon-trained policy speedups do hold for each device.