PRIOR KNOWLEDGE OR SEARCH? A STUDY OF LLM AGENTS IN HARDWARE-AWARE CODE OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM discovery and optimization systems are increasingly applied across domains, implementing a common propose-evaluate-revise loop. Such optimization or discovery progresses via context conditioning on feedback received from an environment. In classical optimization methods, the optimizer's search state is updated, while in LLM-based systems the context is updated but the weights stay fixed. In this work we show that the rich prior encoded in an LLM's weights can be useful in some scenarios, but misleading and detrimental when prior knowledge does not transfer to the task. To demonstrate this, we develop three controlled scenarios, one is optimization over unknown functions, and two low-level code optimization tasks. We find that (1) in function optimization, LLMs act as greedy optimizers and underperform classic methods. (2) In low-level code optimization, performance sharply degrades when generating kernels for uncommon sizes and providing explicit input-size information has no measurable effect. (3) Iterative generation degrades when a less common language is used. Compiled-and-correct rates generally increase for CUDA with iterations but decrease for TVM IR. (4) In several scenarios, classical optimization methods outperform LLM-based systems, even on code optimization tasks. Our results empirically highlight a core limitation of current LLM-based optimization and discovery systems as their performance depends more on pretrained knowledge than on the accumulated context and feedback.

1 INTRODUCTION

Large language models are increasingly deployed as autonomous discovery systems that propose candidate solutions, observe evaluative feedback, and iterate toward novel results. Recent successes span extremal combinatorics (Romera-Paredes et al., 2024), matrix multiplication (Novikov et al., 2025), scientific equation discovery (Shojaee et al., 2025), chemistry (Yang et al., 2025), and GPU kernel optimization (Ouyang et al., 2025; Lange et al., 2025) all sharing the same high level structure, a *propose* $\rightarrow$ *evaluate* $\rightarrow$ *revise* loop. This pattern is structurally identical to Black-Box Optimization (BBO) (Shahriari et al., 2015), with one key difference: a surrogate model updates a posterior over the objective landscape, whereas the LLM encodes its prior in frozen weights and adapts solely through context conditioning. LLMs are effective when this frozen-weight prior, or captured knowledge, supports the target solution space. However, we hypothesize that neither the ability to reason out a solution nor the ability to explore for one will compensate for the lack of a relevant prior in an unfamiliar domain. Although this hypothesis seems simple, it is hard to assess empirically, because prior alignment is not directly observable, and in full agentic pipelines its effect is hard to separate from scaffolding, feedback quality, and search budget. To investigate it, we develop three controlled settings: one involving unknown functions, where task-specific knowledge is limited, and two involving more practical, low-level code (kernel) optimization tasks. In kernel optimization, we control prior density along two axes: (1) representation language, comparing CUDA with TVM intermediate representation (IR), which we reasonably assume is less represented in the models' training data[1], and (2) solution-space familiarity, contrasting standard kernel sizes likely seen during pretraining with small, atypical sizes that are rarely encountered.

[1]This assumption is based on TVM being less frequent in The Stack (Kocetkov et al., 2023), and is further supported by our analysis of the models' output entropy across programming languages in Appendix B.

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.

We summarize our findings and contributions as follows.

1. **Empirical evidence that LLMs perform greedy optimization and require orchestrated exploration to overcome it (§3).** LLMs act as greedy optimizers when optimizing unknown functions, losing to classical algorithms on tasks that require genuine exploration. Combining LLMs with a method that explicitly encourages exploration improves performance.
2. **Weak programming-language priors degrade iterative feedback-loop exploration (§4.1).** The quality of the LLM's prior over the target language determines whether iterative optimization converges or deteriorates. More precisely, the number of correctly compiled kernels generally increases across iterations for CUDA but decreases at each step for TVM IR, where the prior is too weak to support the feedback loop.
3. **Performance degrades sharply when generating kernels for uncommon sizes (§4.1 and §4.2).** Moreover, providing explicit input-size information has no measurable effect as models generate CUDA kernels with generic parameters regardless of the requested size.
4. **Classical methods outperform the LLM where the prior is thin (§3 and §4.1).** At a fixed evaluation budget, CMA-ES and its LLM hybrid Centaur (Ferreira et al., 2026) outperform the plain LLM on the task families that require exploration, and TVM MetaSchedule[2] wins on roughly 70% of kernels when the input sizes move away from the benchmark's canonical regime.

Practical implications. Modern agentic pipelines combine context manipulation, iterative feedback, population-based evolution, and randomized search in ways that make it difficult to attribute performance to any single factor. By working in controlled, minimal settings we isolate these factors and expose failure modes fundamental to any LLM-based discovery system, with direct implications for the design of LLM agents in general and code optimization agents in particular.

Paper roadmap. §2 frames LLM discovery as a form of black-box optimization with a frozen, context-conditioned prior. §3 tests this view with known functions optimization, where no strong code prior is available, and demonstrates that LLMs behave as greedy local refiners. §4 then moves to kernel optimization, where we vary prior alignment through input shape, programming languages, and feedback architecture. Finally, §5 interprets these results and discusses when LLM agents should be paired with explicit search or retrieval rather than used as standalone optimizers. Related work and additional experimental details and results are deferred to the appendix.

2 A Conceptual View of LLM Agents vs. classic BBO

We compare LLM agents and classic black-box optimization (BBO) methods in the same loop. Both propose a candidate, evaluate it, and use the result to choose the next one. For a candidate x in the search space $\mathcal{X}$, at step k the method holds its own parameters θ_k and a context c_k and produces the next candidate,

$$x_{k+1} = q(\theta_k, c_k). \tag{1}$$

Here θ_k are the parameters of the method, the search state for BBO and the model weights for an LLM. Each candidate is evaluated and the environment returns a signal y_{k+1}, which can be a scalar objective value, a compiler error, or a profiling report. The context c_k collects the previous candidates and their signals. In both cases q is a function that returns the next candidate.

Black-Box Optimization. Classic BBO methods keep no explicit context. Each new signal is folded into the parameters by an update rule, so the history reaches the next proposal only through θ_k,

$$x_{k+1} = q(\theta_k), \qquad \theta_{k+1} = U(\theta_k, x_{k+1}, y_{k+1}). \tag{2}$$

The parameters θ_k form a parametric model of the task that is fitted from the observations collected over the iterations of the current run, using an update rule U set by the algorithm. The initial θ_0 carries no task information. The classical example is Bayesian optimization, where θ_k parameterizes a surrogate with mean μ_k and uncertainty σ_k and the next point maximizes an acquisition function such as $\alpha_{\mathrm{UCB}}(x) = \mu_k(x) + \kappa\sigma_k(x)$ (Snoek et al., 2012). Within this same scheme, CMA-ES

[2]TVM MetaSchedule uses XGBoost to score mutated kernel variants and selects a subset to evaluate on hardware; this process is repeated over several iterations.

instead sets θ_k to the mean, step size, and covariance of a Gaussian search distribution (Hansen, 2016).

LLM Agent. The parameters are the same object, but they are frozen, and every new signal is captured in the context,

$$x_{k+1} = q(\theta, c_k), \qquad c_{k+1} = \pi(c_k, x_{k+1}, y_{k+1}), \tag{3}$$

where π is the agent scaffold. Because θ does not change over the iterations of a run, no task-specific quantity is fitted. These frozen weights are what we call *prior knowledge*. They aggregate everything the model acquired during pretraining and post-training, and they alone decide how the context is turned into a candidate. Feedback enters only through c_k, so it can change what the model sees but not how the model reads it. The prior is strong when the task representation and typical solutions were common in training, and weak otherwise. Prompts, history, feedback, tool outputs, and search state belong to c_k and are not prior knowledge.

Practical implication. Classical BBO methods adapt to any task but start from scratch each time. An LLM agent can make good guesses before much feedback is available, but it cannot change how it interprets that feedback. The next sections test where the LLM's prior helps and where it fails.

3 LLMs as Black-Box Optimizers

The model is never told which function or system it is optimizing: it sees only the search bounds and the history of its own proposals and their losses. It does, however, recognize that it is performing optimization.

3.1 Methods and Evaluation for BBO Problems

Problems. We evaluate on four task sets, three two-dimensional and one five-dimensional. The first set contains 100 2d synthetic problems with unknown closed-form **Functions** that have multiple local minima. The second set contains 100 2d problems based on equations from **Physical** dynamical systems. Both task sets were manually constructed. Functions are random compositions of elementary functions, whereas Physical tasks are based on physical laws with time-dependent perturbations. Their construction is described in Appendix C. The third and fourth sets come from the **BBOB** benchmark (Finck et al., 2009) and contain 48 2d and 48 5d problems, respectively.

Optimizers. For each problem, we evaluate **four optimizers** under the same 50-trial budget. *(1) Pure LLM* is a sequential black-box proposer. At each step, it receives only the search history containing previous proposals, corresponding values, and the task prompt, and returns the next candidate point. *(2) CMA-ES* (Hansen, 2016), implemented via Optuna (Akiba et al., 2019) and chosen for its interpretable Gaussian state (mean, step-size, covariance), adapts from objective feedback. *(3) Centaur* Ferreira et al. (2026) enables true co-adaptation by letting the LLM override CMA-ES proposals while CMA-ES learns from them, unlike other hybrids (Liu et al., 2024). *(4) LLM-MCTS* wraps the LLM proposer in a Monte Carlo Tree Search (Coulom, 2006) scaffold. Each expansion selects a node via UCB1 (Kocsis & Szepesvári, 2006), prompts the LLM with the root-to-node path as context, samples k candidate points in parallel, attaches all k as children of the selected node, and back-propagates the minimum child loss through the ancestor chain.

For pure LLM and hybrid methods, we evaluate `gpt-oss-120b`, `DeepSeek-V3.2` (685B), and `DeepSeek-V4-Pro` (1.6T) on the complete set of problems using identical prompts, optimizer configurations, budgets, and metrics. Prompt templates for hybrid and pure LLMs experiments are provided in Appendix D.

Evaluation metrics. We summarize optimizers behavior over fixed budgets through three complementary quantities. *(1) Best step* is the trial at which the best loss is first attained and measures convergence speed. *(2) Coverage* is the fraction of the search domain probed by the query sequence and measures exploration breadth. *(3) Normalized trajectory length* is the ratio of end-to-end displacement to total path length and measures directional consistency, with $L = 1$ for a perfectly linear path and $L \approx 0.14$ for an isotropic random walk at $K = 50$. Full definitions are provided in Appendix E.

3.2 Results for LLMs as BBO Optimizer

Table 1 reports pairwise win counts, average best step, coverage Cov_{50}, and normalized trajectory length L. Figure 2 shows coverage over steps. Together, they reveal three distinct exploration regimes. **(1) LLMs act greedily, which suffices when the objective has many local minima but breaks when the optimum is isolated.** LLMs win on functions, where many local minima make greedy descent toward any plausible basin sufficient. On physical tasks, however, where optima are isolated and exploration is necessary, all three models underperform CMA-ES and Centaur. Low *Coverage* and high L indicate long, locally-confined trajectories rather than broad search. On *BBOB-5D* all three LLMs lose to CMA-ES and Centaur. **(2) MCTS over-explores.** It covers the domain rapidly but exhausts its budget on exploration, rarely exploiting promising regions though we note this is partly a consequence of our hyperparameter choices. **(3) CMA-ES and Centaur have the best balance**. They demonstrate moderate *Coverage* and trajectory length achieving the strongest average performance across tasks. Appendix F presents results on *BBOB-5D* at 75 and 100 trials - overall observations remain the same.

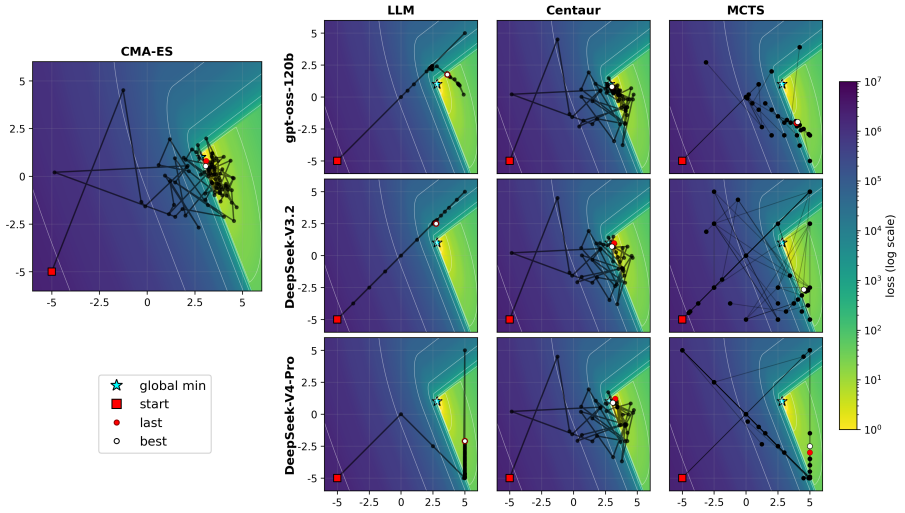

Figure 1: An example of optimization traces. An LLM often follows a greedy, line-like trajectory that either converges quickly or commits to a suboptimal direction. Additional illustrations of two dimensional BBO trajectories are presented in Appendix R.

Table 1: Results for BBO optimization. Entries in the LLM vs optimizer block are pairwise win counts in the form plain-LLM wins vs method wins, with the larger count shown in bold. Each following block presents different metric for each method. The methods are LLM, CMA-ES (CME), Centaur (Cent.), and MCTS. The metrics are average best step, final coverage Cov_{50} (%), and normalized trajectory length L, all definitions are provided in Appendix E.

Model	Task	LLM vs optimizer			Avg. best step				Coverage (%)				L			
		CME	Cent.	MCTS	LLM	CME	Cent.	MCTS	LLM	CME	Cent.	MCTS	LLM	CME	Cent.	MCTS
gpt-oss-120b	Functions	**78** vs 22	**75** vs 25	**83** vs 17	42.8	31.8	37.7	**27.6**	10.9	44.8	45.3	**93.4**	0.72	**0.16**	**0.16**	**0.16**
	Physical	37 vs **63**	29 vs **71**	49 vs **51**	46.7	35.3	44.1	**33.4**	22.4	71.2	71.4	**93.9**	0.67	**0.17**	0.19	**0.17**
	BBOB	23 vs **25**	18 vs **30**	**25** vs 23	40.0	36.2	39.1	**36.1**	11.7	37.1	33.1	**82.1**	0.68	**0.18**	**0.18**	0.27
	BBOB-5D	11 vs **37**	11 vs **37**	19 vs **29**	46.7	**34.7**	43.6	42.3	11.8	41.1	48.5	**69.8**	0.60	**0.11**	0.14	0.32
DeepSeek-V3.2	Functions	**80** vs 20	**81** vs 19	**88** vs 12	40.7	31.8	37.5	**30.4**	24.1	44.8	41.8	**97.1**	0.61	0.16	0.16	**0.15**
	Physical	42 vs **58**	36 vs **64**	**65** vs 35	43.3	35.3	42.6	**30.3**	28.0	71.2	68.8	**99.3**	0.63	0.17	0.21	**0.13**
	BBOB	**28** vs 20	**25** vs 23	**33** vs 15	35.8	36.2	38.0	**31.6**	33.4	37.1	35.7	**97.2**	0.57	0.18	**0.17**	0.19
	BBOB-5D	12 vs **36**	9 vs **39**	23 vs **25**	36.2	34.7	41.3	**24.7**	17.3	41.1	51.8	**97.9**	0.42	**0.11**	0.12	0.14
DeepSeek-V4-Pro	Functions	**89** vs 11	**88** vs 12	**90** vs 10	36.7	**31.8**	42.8	32.2	70.9	44.8	50.0	**98.8**	0.53	0.16	**0.15**	**0.15**
	Physical	40 vs **60**	35 vs **65**	47 vs **53**	47.9	35.3	45.8	**34.3**	14.4	71.2	69.9	**95.8**	0.75	**0.17**	0.20	**0.17**
	BBOB	**26** vs 22	**25** vs 23	**28** vs 20	41.5	36.2	46.0	**35.3**	33.6	37.1	35.1	**70.8**	0.72	**0.18**	**0.18**	0.23
	BBOB-5D	16 vs **32**	11 vs **37**	**32** vs 16	47.2	34.7	43.3	**32.7**	29.8	41.1	49.1	**68.7**	0.56	**0.11**	0.13	0.22
Average	Functions	**82** vs 18	**81** vs 19	**87** vs 13	40.1	31.8	39.3	**30.1**	35.3	44.8	45.7	**96.4**	0.62	0.16	0.16	**0.15**
	Physical	40 vs **60**	33 vs **67**	**54** vs 46	46.0	35.3	44.2	**32.7**	21.6	71.2	70.0	**96.3**	0.68	0.17	0.20	**0.16**
	BBOB	**26** vs 22	23 vs **25**	**29** vs 19	39.1	36.2	41.0	**34.3**	26.2	37.1	34.6	**83.4**	0.66	**0.18**	**0.18**	0.23
	BBOB-5D	13 vs **35**	10 vs **38**	**25** vs 23	43.4	34.7	42.7	**33.2**	19.6	41.1	49.8	**78.8**	0.53	**0.11**	0.13	0.23

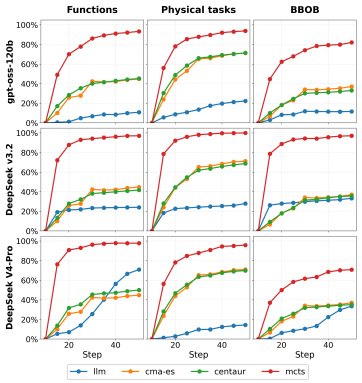

Figure 2: Per-step search-space coverage Cov_k, measured every five steps over 50 trials. The LLM has the lowest coverage. CMA-ES and Centaur occupy a similar mid-range, while LLM-MCTS achieves the highest coverage in every task.

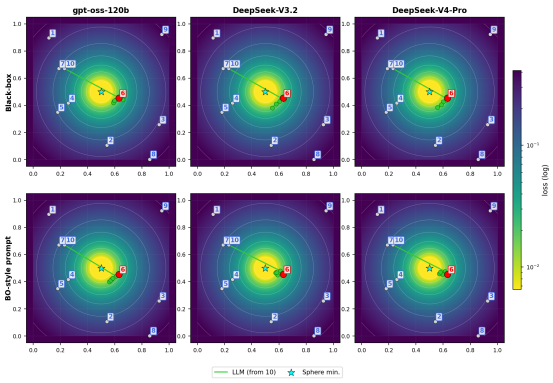

Figure 3: *Is the next step greedy?* Steps 1-10 are uniform-random samples (white labeled circles), the best of these is highlighted in red. The green segment marks future LLM-proposed steps 11-15 and the cyan star is the true minimum. LLMs next proposal is always close to the best in history point regardless prompt.

When prompted to act as Bayesian optimizers, LLMs still behaved greedily (Figure 3). Inspection of the models' chain-of-thought confirms that they attempt to mimic a gradient strategy to justify each successive proposal (Appendix G, Table 5). We further investigate whether earlier steps influence model decisions or whether the model anchors solely on the current best result. To this end, we sampled 10 points on a two-dimensional sphere and then asked each model to propose 5 additional steps. In every case, the next proposal was closest to the current best solution rather than to the most recent step. Results are shown in Figure 3. Experimental protocols and prompt templates are provided in Appendix G.

> With no prior knowledge of the objective, the model falls back on the prior knowledge it does have, knowledge of optimization as a subject, and the strategy it reaches for is local refinement around the best observed point. We speculate this may also be a consequence of the training objective, in which the model is rewarded for predicting the single most likely next token.

4 Kernel Optimization

This section presents three findings on how the LLM's prior knowledge shapes kernel generation:

(i) Iterative feedback-loop optimization degrades rather than improves when prior knowledge is thin (§ 4.1).

(ii) In low-prior settings, classical methods outperform the LLM. TVM MetaSchedule beats the LLM agent on roughly 70% of kernels when tasked with generating kernels for less common input shapes (§ 4.1).

(iii) Generated kernels reuse a generic set of kernel parameters, even when the prompt explicitly specifies the input shape. However, this generic set of parameters is not optimal for all input shapes (§ 4.2).

4.1 Iterative Hardware-Aware Kernel Optimization with Feedback

We evaluate LLM-optimized kernels in terms of correctness and latency across two code representations (CUDA and TVM IR), two optimization pipelines, and two input-size regimes. We first reasonably assume that TVM IR is less represented than CUDA in the models' training data. Frequency in The Stack (Kocetkov et al., 2023) is a common proxy for how well a language is represented in pretraining data (Cassano et al., 2024), and Figure 4 shows that TVM is indeed less frequent than CUDA there. However, because public-code counts are only a proxy for what a model has seen, particularly for models with undisclosed training data, we also measure the token-level

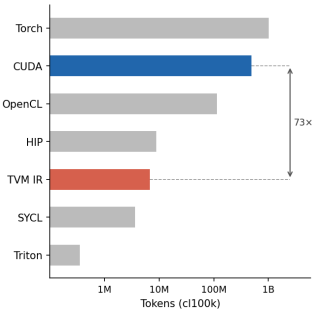

Figure 4: Token counts for GPU-related languages in The Stack. The x-axis is logarithmic.

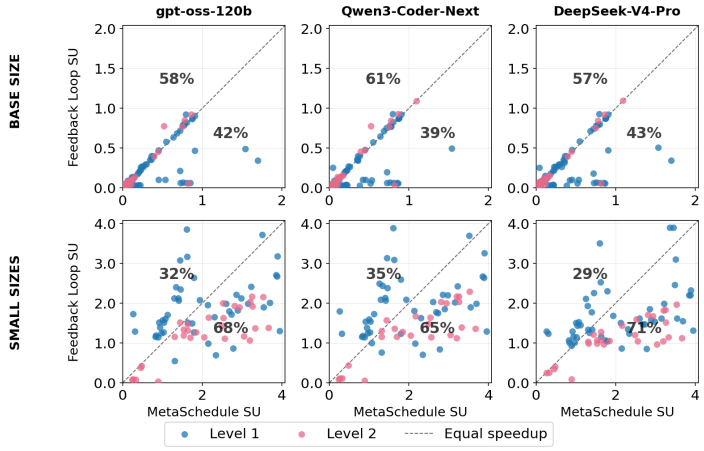

Figure 5: Per-kernel speedup of TVM MetaSchedule against the LLM-driven TVM IR Feedback Loop. Points above the diagonal are kernels where the LLM-generated kernels are faster, and percentages show the fraction of kernels in each triangle. When input sizes change, MetaSchedule wins for most kernels and models.

entropy of the models' own generations and find that it is higher for TVM IR than for CUDA across all operations and models we tested (Appendix B).

Procedure: For CUDA, the model translates a PyTorch reference to a CUDA kernel. For TVM IR, the reference is an *unscheduled* TVM-TIR program derived from the same PyTorch model via automated conversion (ONNX→Relax→TIR, with schedule annotations stripped), so the computation is already correctly implemented. This allows us to have two computationally equivalent tasks but represented differently. The pipeline is described in Appendix L. The TVM agent uses injected syntax guidance needed to reliably generate valid code [3]. Full prompts are in Appendix M.

Size regimes. We evaluate two input-size regimes using problems from KernelBench (Ouyang et al., 2025). The *Base* (benchmark-native) regime uses the original shapes and the *Small* regime uses substantially reduced sizes (e.g. matrix multiplication shrinks from M=2048, K=8192, N=4096 to M=128, K=512, N=256). More examples are provided in Appendix O. All experiments cover two kernel difficulty levels of KernelBench with 100 kernels per level for each size regime.

Baselines. `torch.compile` is the main performance reference, all speedups are reported relative to it. TVM MetaSchedule is the primary non-LLM baseline for kernel generation, which implements the same iterative feedback-loop, it updates XGBoost as its state to score mutated kernel variants to propose next candidates. We use default settings of TVM MetaSchedule for all the experiments.

Agent pipelines. The *Sampling Agent* generates five candidates independently, converts each outcome into structured feedback, and conditions a sixth candidate on all five reference-candidate pairs. The *Feedback Loop* pairs an *advisor/diagnoser* agents with a *coder* over up to five sequential iterations, the advisor recommends further optimizations when the kernel is correct, or the diagnoser prescribes a fix when it is not. Pipeline diagrams are provided in Appendix P. The CUDA advisor/diagnoser prompts, structured-feedback format, sampling post-processing prompt, and generation settings are reported in Appendix N.

Models. The feedback-loop experiment uses `gpt-oss-120b`, `Qwen3-Coder-Next` (80B)[4], and `DeepSeek-V4-Pro` (1.6T). Non-code-generation calls are sampled at temperature 0.7 and code generation calls at 0.1. For `gpt-oss-120b` we use medium reasoning effort. All kernel generation experiments were conducted on a server with 8 NVIDIA A100-SXM4-80GB GPUs, CUDA 12.6, PyTorch 2.9.0, and an AMD EPYC 7513 32-Core CPU (2.0 TiB RAM).

[3] Our initial experiments demonstrated significantly lower performance without syntax guidance for TVM.

[4] In contrast with function optimization section we replaced DeepSeek-V3.2 with Qwen3-Coder-Next because the former consistently failed to generate valid TVM IR.

Results for Iterative Kernel Optimization with Feedback. **(1) Feedback behaves differently across the CUDA and TVM IR settings.** Figure 6 shows that, from the first to the fifth iteration, compiled-and-correct rates generally increase for CUDA across all three models. TVM IR behaves in the opposite way: compiled-and-correct rates decrease, while changes in correct-and-faster rates are mixed. Iterative feedback effectively steers proposals when the model has a dense prior over the target representation, but it destabilizes generation when that prior is sparse. Importantly, TVM IR starts from a semantically stronger reference, namely a valid unscheduled program, yet its pass rate still declines. This highlights that the effect is driven by prior density rather than reference quality.

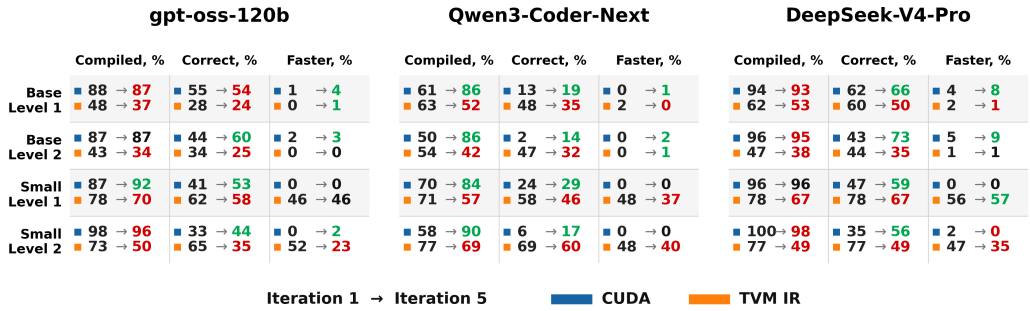

Figure 6: Feedback-loop results for iterations 1 to 5. Colors at 5th iteration **1** → **5** denote **improvement**, **degradation**, or **no change**. Metrics are cumulative, not additive, *faster* denotes the fraction of kernels that are correct and faster than `torch.compile`. Kernel difficulty levels 1–2 are original KernelBench (Ouyang et al., 2025). *Base* is the original KernelBench configuration, and *Small* is a configuration with reduced input sizes. Intermediate iterations are presented in Appendix Q.

(2) Input size as a distribution shift. Figure 5 shows that in the *Base* regime, the LLM-driven TVM IR feedback loop is roughly competitive with MetaSchedule. In the *Small* regime, the pattern reverses, MetaSchedule wins on 65–71% of kernels, depending on the model, with the effect strongest on Level 2 kernels. We hypothesize that small, atypical shapes are harder for the LLM because its proposals carry parameter biases calibrated to more common shapes, whereas MetaSchedule has no such prior and explores the space freely.

(3) Sequential feedback is better than parallel sampling. Figure 7 shows iterative gain relative to the iteration-1 baseline. Feedback loop based agent consistently outperforms the sampling agent at matched budget. The two converge only with CUDA in the *Base* regime - densest-prior corner, elsewhere the gap is substantial. Two factors likely explain sampling's underperformance, an inflated context window and the fact that many small iterative corrections are better in general rather than one big update.

Relative gains are greater for TVM IR, but only because CUDA starts from a stronger LLM-generated baseline with less headroom to improve. This does not contradict the absolute degradation in Figure 6, since the improved and degraded kernel subsets are asymmetric.

4.2 Shape Bias in Kernel Generation

Here we ask whether the model conditions on the requested input size at all. To test this, we prompt models to generate CUDA kernels and specify the input shapes these kernels will be applied to. We construct a shape grid for three kernels [5] spanning 10 input sizes from small to large. Each prompt includes a Python reference implementation with concrete input sizes. We evaluate two instruction regimes: *implicit* (sizes stated once in the task description) and *explicit* (the same, plus a direct instruction to attend to the listed sizes). We generate 20 kernels per grid point at different temperatures $T \in \{0.1, 0.5, 0.8\}$, recording the pass rate and the dominant tiling parameters in the generated kernels. Full prompt templates are in Appendix H.

[5] (1) Softmax, (2) matrix multiplication with a transposed input, and (3) a convolution fused with batch normalization, Tanh, max pooling, and group normalization.

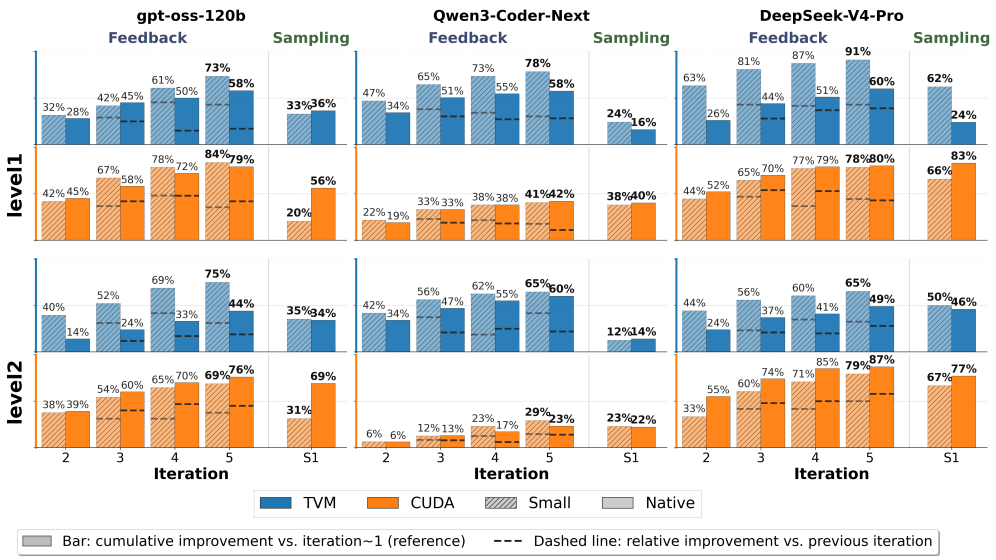

Figure 7: Iterative improvement relative to the first iteration under two agent structures (feedback loop vs. sampling). Feedback bars at iterations 2-5 show the cumulative fraction of kernels improved over iteration 1. Dashed segments show per-step improvement. The Sampling column (S1) shows the single post-aggregation candidate.

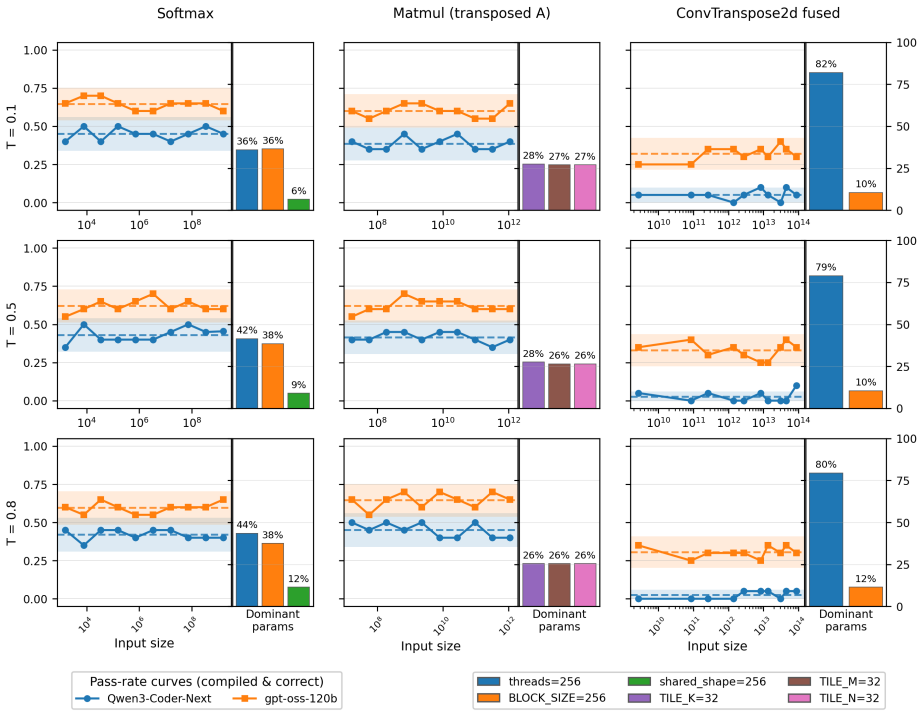

Figure 8: Pass rate (left) and dominant kernel parameters (right) across shape grids for three kernels, evaluated at different temperatures. Pass rate is averaged over the explicit and implicit prompt conditions, as both yield indistinguishable curves. Flat pass rate means that generated kernels produce correct results regardless required input shapes. Parameter values are not constrained in the prompts, yet the models repeatedly generate the same values. For Matmul, three distinct values cover up to 78% of generated kernels.

Results for Shape-Bias Analysis. The left panel of Figure 8 shows pass-rate curves that are essentially flat across the entire size grid for both models [6] and at all three temperatures, and the explicit and implicit prompt conditions produce nearly identical curves. The right panel shows the distribution of the parameters that control hardware performance.[7] Although we did not restrict these parameter values, a small subset of values dominates the generated kernels, three distinct values cover up to 78% of kernels. This collapse occurs regardless of input size or temperature (see Appendix I for detailed results). Together with the flat pass rate, this indicates that the models produce essentially the same kernel configurations regardless of input size.

One might argue that this is acceptable as long as the kernels remain correct across shapes. However, Appendix J shows that these generic parameters are not universally optimal. The best parameters depend on the input size, and the model's fixed choices incur slowdowns of up to $1.76\times$.

We further ask whether kernel search is greedy, as in function optimization. To test this, we fix the kernel code and optimize only the parameter values. Appendix K shows that LLMs explore a much narrower range of parameter values than Optuna-TPE, the same pattern we observed in black-box optimization (§3). Moreover, the parameters found by Optuna-TPE lead to faster kernels.

> Across two experiments, LLM optimization is less reliable when the task is formulated in TVM IR or the input shape is atypical. Iterative feedback helps in familiar domain - CUDA but hurts otherwise - TVM IR. MetaSchedule overtakes the LLM agent once froze prior is weak and is not updated. Kernel generation reuses common parameter regardless tasked input sizes. These observations are consistent with a frozen-weight-prior interpretation in § 2.

5 Discussion

Prior alignment may shape context use. Here, prior knowledge means information and behavioral biases encoded in frozen weights before the optimization episode, including effects of pretraining and post-training. Prompts, history, feedback, and search state instead belong to the context or pipeline (§ 2). The greedy local search in BBO (§ 3.2) and the mode collapse in shape-conditioned kernel generation (§ 4) are consistent with the model favoring contextual information aligned with its frozen-weight prior. In BBO, the next decision anchors to the best current value. In kernel generation, the model reuses common parameter modes without mapping the requested shape into specialized tile, vector-width, or unrolling choices. We treat prior alignment as an empirical interpretation.

LLM exploration can be improved in the following ways. (1) Either by adjusting prior via domain-specific RL (Dai et al., 2026), (2) retrieving examples with the informative patterns at inference time and (3) by forcing exploration externally through agentic scaffolding.

Limitations and Broader Impact. Our study is limited to a few model families and standard benchmarks. The Stack counts are only a proxy for representation prevalence in public code. The evaluated models' actual pretraining and post-training mixtures are unknown, so Figure 4 does not directly measure their training exposure. Moreover, CUDA and TVM IR differ in task formulation as well as representation, preventing a causal attribution to prior familiarity alone. Better kernel generation could reduce energy costs, but auto-generated code demands rigorous validation before production use.

6 Conclusion

This work argues that the central variable in agentic optimization is not whether feedback exists, but whether feedback can be absorbed by the model's prior. Across BBO, shape-conditioning, and hardware optimization, we observe a common pattern. LLMs are strong prior exploiters and weak open-ended searchers. From this view, three takeaways follow.

First, feedback is representation-conditional. It is beneficial in dense-prior domains and can destabilize in sparse-prior ones. Second, sequential conditioning on concrete external signals is more effective than one-shot aggregation of many sampled trajectories at similar budget. Third, robust optimization pipelines should route adaptively. Use LLM refinement where prior coverage is dense, and rely on task-adaptive search when distribution shift or sparse prior makes language-only adaptation unreliable.

[6] `DeepSeek-V4-Pro` was not evaluated in this probe because the full shape grid was computationally too expensive for a 1.6T model.

[7] Block size, tile size, vector width, and unroll factor.

AI USE STATEMENT

Generative AI was used to prepare figures and generate optimization candidates and kernel code in the reported experiments. Experimental outputs were evaluated using objective functions, correctness checks, and runtime measurements, as described in the paper.

REPRODUCIBILITY STATEMENT

Sections 3 and 4.1 describe the experimental setups, models, evaluation budgets, and hardware. The appendices provide prompt templates (Appendices D and H), metric definitions (Appendix E), and the TVM reference construction pipeline (Appendix L).

REFERENCES

Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 2623–2631. ACM, 2019. doi: 10.1145/3292500.3330701.

Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Anders Freeman, Carolyn Jane Anderson, Molly Q Feldman, Michael Greenberg, Abhinav Jangda, and Arjun Guha. Knowledge transfer from high-resource to low-resource programming languages for code LLMs. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):677–708, October 2024. doi: 10.1145/3689735. URL https://doi.org/10.1145/3689735.

Rémi Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In *Computers and Games (CG)*, pp. 72–83. Springer, 2006. doi: 10.1007/978-3-540-75538-8_7.

Weinan Dai, Hanlin Wu, Qiying Yu, Huan-Ang Gao, Jiahao Li, Chengquan Jiang, Weiqiang Lou, Yufan Song, Hongli Yu, Jiaze Chen, Wei-Ying Ma, Ya-Qin Zhang, Jingjing Liu, Mingxuan Wang, Xin Liu, and Hao Zhou. CUDA agent: Large-scale agentic RL for high-performance CUDA kernel generation, 2026.

Fabio Ferreira, Lucca Wobbe, Arjun Krishnakumar, Frank Hutter, and Arber Zela. Can llms beat classical hyperparameter optimization algorithms? A study on autoresearch, 2026.

Steffen Finck, Nikolaus Hansen, Raymond Ros, and Anne Auger. Real-parameter black-box optimization benchmarking 2009: Noiseless functions definitions. Technical Report RR-6829, INRIA, 2009.

Xiangming Gu, Soham De, Michalis Titsias, Larisa Markeeva, Petar Veličković, and Razvan Pascanu. The illusion of stochasticity in LLMs, 2026.

Nikolaus Hansen. The CMA evolution strategy: A tutorial, 2016.

Sen Huang, Kaixiang Yang, Sheng Qi, and Rui Wang. When large language model meets optimization. *Swarm and Evolutionary Computation*, 90:101663, 2024. doi: 10.1016/J.SWEVO.2024.101663.

Valentin Khrulkov, Andrey V. Galichin, Denis Bashkirov, Dmitry Vinichenko, Oleg Travkin, Roman Alferov, Andrey Kuznetsov, and Ivan V. Oseledets. Gigaevo: An open source optimization framework powered by llms and evolution algorithms, 2025.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Yacine Jernite, Margaret Mitchell, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. The stack: 3 TB of permissively licensed source code. *Transactions on Machine Learning Research (TMLR)*, 2023.

Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *European Conference on Machine Learning (ECML)*, pp. 282–293. Springer, 2006. doi: 10.1007/11871842_29.

Robert T. Lange, Yingtao Tian, and Yujin Tang. Evolution transformer: In-context evolutionary optimization. In *Genetic and Evolutionary Computation Conference (GECCO)*, pp. 575–578. ACM, 2024. doi: 10.1145/3638530.3654393.

Robert Tjarko Lange, Qi Sun, Aaditya Prasad, Maxence Faldor, Yujin Tang, and David Ha. Towards robust agentic CUDA kernel benchmarking, verification, and optimization, 2025.

Gang Liao, Hongsen Qin, Ying Wang, Alicia Golden, Michael Kuchnik, Yavuz Yetim, Jia Jiunn Ang, Chunli Fu, Yihan He, Samuel Hsia, Zewei Jiang, Dianshi Li, Uladzimir Pashkevich, Varna Puvvada, Feng Shi, Matt Steiner, Ruichao Xiao, Nathan Yan, Xiayu Yu, Zhou Fang, Roman Levenstein, Kunming Ho, Haishan Zhu, Alec Hammond, Richard Li, Ajit Mathews, Kaustubh Gondkar, Abdul Zainul-Abedin, Ketan Singh, Hongtao Yu, Wenyuan Chi, Barney Huang, Sean Zhang, Noah Weller, Zach Marine, Wyatt Cook, Carole-Jean Wu, and Gaoxiang Liu. KernelEvolve: Scaling agentic kernel coding for heterogeneous AI accelerators at Meta, 2025.

Tennison Liu, Nicolás Astorga, Nabeel Seedat, and Mihaela van der Schaar. Large language models to enhance bayesian optimization. In *The Twelfth International Conference on Learning Representations*, 2024.

Alexander Novikov, Ngân Vu, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. Alphaevolve: A coding agent for scientific and algorithmic discovery, 2025.

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

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625:468–475, 2024. doi: 10.1038/S41586-023-06924-6.

Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1): 148–175, 2015.

Parshin Shojaee, Kazem Meidani, Shashank Gupta, Amir Barati Farimani, and Chandan K. Reddy. LLM-SR: Scientific equation discovery via programming with large language models. In *International Conference on Learning Representations (ICLR)*, 2025.

Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 2960–2968, 2012.

Kaya Stechly, Karthik Valmeekam, and Subbarao Kambhampati. Chain of thoughtlessness? An analysis of CoT in planning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Yiyou Sun, Shawn Hu, Georgia Zhou, Ken Zheng, Hannaneh Hajishirzi, Nouha Dziri, and Dawn Song. OMEGA: can llms reason outside the box in math? evaluating exploratory, compositional, and transformative generalization, 2025.

Ziyang Xiao, Jingrong Xie, Lilin Xu, Shisi Guan, Jingyan Zhu, Xiongwei Han, Xiaojin Fu, WingYin Yu, Han Wu, Wei Shi, Qingcan Kang, Jiahui Duan, Tao Zhong, Mingxuan Yuan, Jia Zeng, Yuan Wang, Gang Chen, and Dongxiang Zhang. A survey of optimization modeling meets LLMs: Progress and future directions. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 10742–10750. ijcai.org, 2025. doi: 10.24963/IJCAI.2025/1192.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *International Conference on Learning Representations (ICLR)*, 2024.

Zonglin Yang, Wanhao Liu, Ben Gao, Tong Xie, Yuqiang Li, Wanli Ouyang, Soujanya Poria, Erik Cambria, and Dongzhan Zhou. MOOSE-Chem: Large language models for rediscovering unseen chemistry scientific hypotheses. In *International Conference on Learning Representations (ICLR)*, 2025.

Xinhao Zhang, Xi Chen, François Portet, and Maxime Peyrard. What makes an LLM a good optimizer? A trajectory analysis of LLM-guided evolutionary search, 2026.

Chengshuai Zhao, Zhen Tan, Pingchuan Ma, Dawei Li, Bohan Jiang, Yancheng Wang, Yingzhen Yang, and Huan Liu. Is chain-of-thought reasoning of llms a mirage? A data distribution lens, 2025.

Zihan Zheng, Zerui Cheng, Zeyu Shen, Shang Zhou, Kaiyuan Liu, Hansen He, Dongruixuan Li, Stanley Wei, Hangyi Hao, Jianzhu Yao, Peiyao Sheng, Zixuan Wang, Wenhao Chai, Aleksandra Korolova, Peter Henderson, Sanjeev Arora, Pramod Viswanath, Jingbo Shang, and Saining Xie. Livecodebench pro: How do olympiad medalists judge llms in competitive programming?, 2025.

APPENDIX

The appendix is organized as follows. Appendix A surveys related work on LLM reasoning, LLM-based optimization, and LLM-based kernel generation. Appendix B reports the token-level entropy experiment across code representations. Appendix C describes how the Functions and Physical task sets were constructed. Appendix D lists the exact prompt templates used in the synthetic BBO comparison. Appendix E defines the evaluation metrics for the pure BBO experiments. Appendix F repeats the BBOB-5D comparison at larger evaluation budgets. Appendix G details the experimental protocol for the greedy-gradient probe. Appendix H provides the prompt templates for shape-conditioned zero-shot generation. Appendix I tabulates the dominant zero-shot parameter modes across temperatures. Appendix J verifies that runtime-optimal kernel parameters depend on input size and that the LLM's repeated canonical settings are not universal defaults. Appendix K reports a separate controlled experiment comparing greedy parameter exploration by the LLM with Optuna-TPE on fixed CUDA kernels. Appendix L describes the pipeline for constructing the unscheduled TVM-TIR reference module. Appendix M reproduces the analysis, code, and summary prompts of the TVM agent. Appendix N reproduces the CUDA advisor/diagnoser prompts, structured-feedback format, and sampling post-processing prompt, and reports generation settings. Appendix O lists tensor shapes for five example tasks in the Base and Small regimes. Appendix P illustrates the two agentic pipeline architectures via flowcharts and a summary table. Appendix Q expands the feedback-loop validity figure to all five iterations. Appendix R provides per-task BBOB optimization trace plots.

A RELATED WORK

We survey prior work across three threads directly relevant to our study, namely (i) the limits of LLM reasoning under distributional shift, (ii) LLMs as optimization operators and search primitives, and (iii) LLM-based approaches to GPU kernel generation.

LLM Reasoning. A growing body of work challenges the assumption that frontier LLMs possess robust, generalizable reasoning, converging instead on a picture of brittle pattern matching that degrades rapidly under distributional shift. Stechly et al. (2024) provide an early and influential demonstration of this through careful experimentation in the Blocksworld planning domain. Varying both the generality of in-context examples and the complexity of test problems, they find that Chain of Thought (CoT) prompting yields meaningful gains only when prompts are narrowly tailored to the specific problem class and that even these gains collapse quickly as problem size grows beyond what was demonstrated. Zhao et al. (2025) extend and formalize this critique through an explicit distributional lens. Introducing a controlled training environment, they demonstrate that CoT behavior reflects a structured bias learned from in-distribution data, whose effectiveness is fundamentally bounded by the degree of mismatch between training and test distributions. Sun et al. (2025) evaluates out-of-distribution generalization along three axes exploratory, compositional, and transformative using programmatically generated problem templates with adjustable complexity. Evaluations of frontier models reveal sharp, monotonic accuracy degradation as complexity increases. Crucially, reinforcement learning fine-tuning on low-complexity data yields strong in-distribution gains but provides only limited and rapidly plateauing improvements out-of-distribution, with transformative generalization showing virtually no improvement at all. In the domain of competitive programming, Zheng et al. (2025) corroborate these findings using a contamination-free benchmark of 584 problems drawn from Codeforces, ICPC, and IOI, continuously updated and annotated by Olympiad medalists. Even the best reasoning models achieve only 53% accuracy on medium-difficulty problems and 0% on hard problems, with failures concentrated on observation-heavy problems that require novel insights rather than the recall of known techniques. Fine-grained error analysis further reveals that conceptual failures dominate over implementation bugs, and that tool augmentation accounts for a substantial share of performance gains that do not reflect native reasoning ability. Gu et al. (2026) question whether LLMs can stochastically explore environments by evaluating their ability to perform independent and sequential sampling of random numbers, finding that both approaches fail standard independence tests. However, the authors show that models can reliably *convert* externally provided random seeds to target distributions suggesting that the failure lies not in distributional understanding but in autonomous random generation. This knowing-doing gap offers a partial explanation for why tool augmentation so consistently boosts LLM performance in agentic settings.

LLM-Based Optimization. A complementary line of research reframes LLMs from problem-solvers to search operators, algorithm designers, and formal modelers within optimization loops (Huang et al., 2024). Yang et al. (2024) pioneered the use of LLMs as iterative optimizers via Optimization by PROmpting (OPRO), while subsequent work has expanded this to using LLMs for translating natural language directly into rigorous mathematical optimization models (Xiao et al., 2025). When deployed as evolutionary search operators, trajectory analyses reveal that effective LLM optimizers diverge significantly from general reasoning behaviors. Zhang et al. (2026) find that strong optimizers act as *local refiners*, producing frequent incremental improvements while maintaining a localized search region, whereas high novelty often leads to semantic drift without fitness gains. This behavioral nuance is reflected in empirical benchmarks. Ferreira et al. (2026) show that classical algorithms like CMA-ES consistently outperform pure LLM agents in fixed hyperparameter spaces, though hybrid approaches that share the classical optimizer's internal state (e.g., their proposed **Centaur** framework) achieve state-of-the-art results by letting the LLM apply domain knowledge to well-guided proposals. Architecturally, this fusion is formalized by Lange et al. (2024), who introduce the Evolution Transformer to distill evolutionary strategy updates into a causal attention mechanism, offering a structured alternative to purely text-based LLM search.

LLM-Based Kernel Generation. A recurring pattern is the pairing of LLMs with explicit exploration mechanisms. A notable production-oriented system is **KernelEvolve** (Liao et al., 2025), developed for generating optimized kernels across heterogeneous accelerators (NVIDIA, AMD, MTIA). It integrates tree search, agentic retrieval, knowledge base of hardware constraints, and a profiling feedback. However, the source code is proprietary, and the paper includes no ablation studies. Consequently, it remains unclear which components critically contribute to the reported gains. An open-source counterpart is KernelEvo, built on the GigaEvo evolutionary algorithm framework (Khrulkov et al., 2025). It provides an implementation of LLM-guided evolutionary kernel generation. Yet it lacks a peer-reviewed paper, systematic evaluation and any ablation studies. **CUDA Agent** (Dai et al., 2026) develops a large-scale reinforcement learning system for CUDA kernel generation, using a synthetic dataset of fused operators, a structured agent toolset, and PPO with multi-stage warm-up. The system reports strong results on KernelBench. However, the work is limited to CUDA and lacks comparison with classical black-box optimizers (e.g., TVM MetaSchedule). Broadly, three strategies dominate, namely (i) tree search, (ii) population-based evolutionary diversification, and (iii) reinforcement learning that adapts the model weights themselves rather than only the context.

B Token-Level Entropy Across Code Representations

What this measures. Figure 4 counts public code, which is an indirect estimate of how much of each representation a model saw. Here we test whether the gap also appears in the models' own output distributions during code generation. Higher token-level entropy means probability is spread over more possible next tokens and therefore indicates less confident generation.

Setup. We evaluate `gpt-oss-120b` and `Qwen3-Coder-Next` on seven kernel operations, namely ReLU, Softmax, Matmul, LayerNorm, Conv2d, cross-entropy and KL divergence. Each operation is generated in six target representations, CUDA, OpenCL, HIP, TVM IR, SYCL and Triton. All conditions use the same zero-shot system instruction, which asks the model to act as an expert GPU kernel engineer and return only code. The user prompt specifies the operation and the target representation, and across the six matched versions only the requested target representation changes.

Entropy computation. We record the next-token distribution produced at every generated code position. For vocabulary $\mathcal{V}$, the entropy at position t is

$$h_t = -\sum_{v \in \mathcal{V}} p_t(v) \log p_t(v), \tag{4}$$

where $p_t(v)$ is the probability assigned to token v. For an output of T code tokens, we compute the mean entropy $\bar{h} = \frac{1}{T} \sum_{t=1}^{T} h_t$. Table 2 reports this mean for every operation and representation, and Figure 9 summarises each representation as a mean and standard deviation over the seven operations.

Table 2: Mean token-level entropy for every kernel operation and code representation. TVM IR is higher than CUDA for all seven operations under both models.

Model	Operation	CUDA	OpenCL	HIP	TVM IR	SYCL	Triton
`gpt-oss-120b`	ReLU	0.044	0.048	0.055	0.121	0.062	0.066
	Softmax	0.071	0.069	0.081	0.076	0.083	0.096
	Matmul	0.049	0.073	0.047	0.067	0.054	0.069
	LayerNorm	0.059	0.114	0.054	0.094	0.055	0.097
	Conv2d	0.059	0.046	0.047	0.072	0.057	0.048
	Cross-entropy	0.067	0.112	0.054	0.121	0.087	0.093
	KL divergence	0.055	0.093	0.060	0.144	0.086	0.108
`Qwen3-Coder-Next`	ReLU	0.112	0.090	0.099	0.147	0.104	0.129
	Softmax	0.102	0.084	0.116	0.106	0.127	0.168
	Matmul	0.064	0.059	0.064	0.206	0.117	0.083
	LayerNorm	0.107	0.096	0.130	0.159	0.112	0.147
	Conv2d	0.101	0.063	0.176	0.184	0.066	0.100
	Cross-entropy	0.105	0.103	0.115	0.156	0.106	0.243
	KL divergence	0.115	0.146	0.108	0.162	0.103	0.139

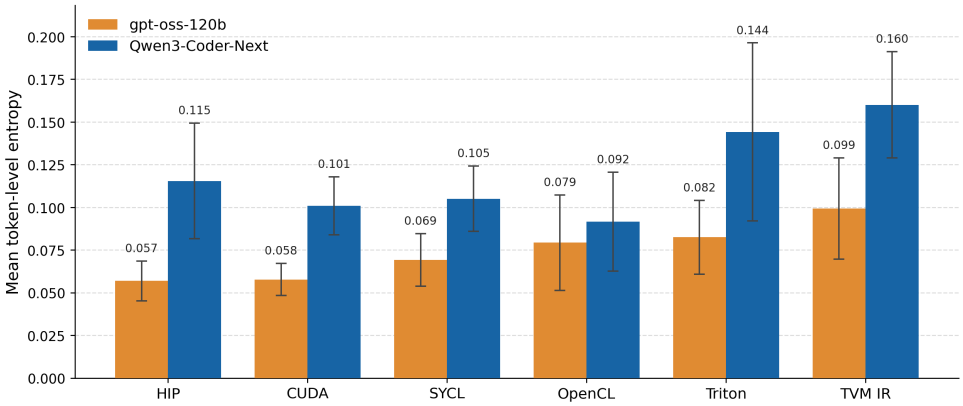

Figure 9: Mean token-level entropy per code representation, averaged over the seven kernel operations of Table 2. Bars are the operation means and error bars the sample standard deviation across the seven operations. Orange is `gpt-oss-120b` and blue is `Qwen3-Coder-Next`. Representations are ordered by the `gpt-oss-120b` mean. TVM IR is the highest of the six representations for both models and CUDA is among the lowest.

Results. TVM IR has higher entropy than CUDA for every evaluated operation under both models. Averaged across operations, entropy increases from 0.058 for CUDA to 0.099 for TVM IR with `gpt-oss-120b`, a 1.71× difference, and from 0.101 to 0.160 with `Qwen3-Coder-Next`, a 1.58× difference. TVM IR also has the highest mean among all six representations for both models, whereas CUDA is among the lowest. The consistent CUDA–TVM IR gap shows that both models generate TVM IR less confidently than CUDA under matched prompts. This supports the interpretation that TVM IR is a less familiar representation for the evaluated models. The experiment establishes a generation-level confidence difference, but does not by itself identify the source of that difference.

C Construction of the Functions and Physical Task Sets

Functions. Each of the 100 tasks minimizes $L(x, y) = |F(x, y)|$ over $(x, y) \in [0, 1]^2$. With a fixed random seed, F is constructed by composing elementary shifted power and trigonometric terms. Each expression combines polynomial components in x and y, cross-variable power terms, and sine or cosine components whose arguments depend on one or both variables. Shifts are sampled uniformly from $[0.2, 0.8]$, polynomial degrees range from 2 to 5, trigonometric frequencies are integers from 4

to 9, and all component weights are randomly sampled. A candidate expression is retained only if it changes with both variables, is finite and nonconstant on 11 validation points.

Physical tasks. The 100 tasks are based on standard response equations from five physical systems, with 20 instances per system, namely $g(t;k,b) = kt + b$ for a spring, $g(t;k,b) = k\sin(bt)$ for a pendulum, $g(t;k,b) = k\sin(2\pi bt)$ for alternating current, $g(t;k,b) = k\exp(-bt)\cos(t)$ for a damped oscillator, and $g(t;k,b) = k[1 - \exp(-t/(b + 0.01))]$ for an RC circuit. For each instance, hidden parameters (k^*, b^*) are sampled from the predefined search range and the target trajectory is evaluated at 20 uniformly spaced time points $t_j = j/19$. We add structured time-dependent noise generated from random combinations of degree-1–3 polynomial and sine/cosine functions of time, with the noise contribution scaled by 0.1. Thus, the target values are $z_j = g(t_j;k^*,b^*) + 0.1p(t_j;k^*,b^*)$. For a proposed (k,b), the same task-specific perturbation is used to obtain predictions $\hat{z}_j = g(t_j;k,b) + 0.1p(t_j;k,b)$, and the loss is the root-mean-square error $L(k,b) = \sqrt{\frac{1}{20}\sum_{j=0}^{19}(\hat{z}_j - z_j)^2}$.

D Prompt Templates for the Synthetic BBO Comparison

This section reports the exact prompt templates used when asking the LLM to act as a black-box optimizer in the synthetic comparison from Section 3.2.

Two-dimensional function optimization.

```
You are optimizing an unknown black-box function.
Goal: suggest x and y that minimize this function (smaller is better).
Search bounds: x and y must be in [0, 1].
Trial number: <trial_num>.
Current best loss: <best_loss>.
History of all previous attempts (step | x | y | loss):
<history_lines>
Important: do NOT repeat any previous (x, y) pair from history.
Always propose a new point that was not tried before.
Return ONLY JSON in this exact format: {"x": <float>, "y": <float>}.
```

Physical-system parameter optimization.

```
You are optimizing physical system parameters.
Goal: suggest k and b that minimize loss (smaller is better).
Search bounds: k in [<k_min>, <k_max>], b in [<b_min>, <b_max>].
Trial number: <trial_num>.
Current best loss: <best_loss>.
History of all previous attempts (step | k | b | loss):
<history_lines>
Important: do NOT repeat any previous (k, b) pair from history.
Always propose new parameters that were not tried before.
Return ONLY JSON in this exact format: {"k": <float>, "b": <float>}.
```

BBOB function optimization.

```
You are optimizing an unknown black-box function (BBOB benchmark).
Goal: suggest parameters that minimize this function.
Search bounds: <bounds_str>.
Trial number: <trial_num>.
Current best loss: <best_loss>.
History of all previous attempts (step | x0 | x1 | loss):
<history_lines>
Important: do NOT repeat any previous parameter combination.
Always propose new parameters.
Return ONLY JSON: <keys_desc>.
```

In the two-dimensional BBOB runs used in the main experiments, `<bounds_str>` has the form `x0 in [<lower_0>, <upper_0>], x1 in [<lower_1>, <upper_1>]`, and `<keys_desc>` is `{"x0": <float>, "x1": <float>}`.

E Evaluation Metrics for Pure BBO Optimization Problems

Each task τ fixes a bounded feasible region $\mathcal{X}_\tau \subset \mathbb{R}^d$ (e.g. $[0,1]^2$ for the function family or the task-specific box for BBOB and parameter identification) that all four optimization settings see and must sample from. For each run we record the sequence of queried points $\{x_1, \dots, x_K\} \subset \mathcal{X}_\tau$ (with $K = 50$) and summarize behavior through three quantities.

Best step $\tau^\star = \arg\min_{k \in \{1,\dots,K\}} y_k$ is the trial index at which the best loss is first attained, averaged over tasks. It measures how quickly an optimizer locks onto its final solution within the budget.

Coverage reports how broadly the domain has been probed. Let $\bar{x}_k$ be the centroid of the first k queries and r_k the distance from $\bar{x}_k$ to the farthest of them,

$$\bar{x}_k \;=\; \frac{1}{k}\sum_{i=1}^{k} x_i, \qquad r_k \;=\; \max_{1 \le i \le k} \|x_i - \bar{x}_k\|_2. \tag{5}$$

For a d-dimensional search space, coverage is the volume of the resulting bounding d-ball divided by the volume of the search domain. The volume of a radius-r_k ball in $\mathbb{R}^d$ is

$$V_d(r_k) \;=\; \frac{\pi^{d/2}}{\Gamma\!\left(\frac{d}{2}+1\right)} r_k^d, \qquad \mathrm{Cov}_k \;=\; \frac{V_d(r_k)}{\mathrm{Vol}_d(\mathcal{X}_\tau)}. \tag{6}$$

For the two-dimensional experiments, the numerator is $V_2(r_k) = \pi r_k^2$. For BBOB-5D, it is $V_5(r_k) = \frac{8\pi^2}{15} r_k^5$. Reported as a percentage, Cov_k indicates roughly how large a fraction of the search domain the optimizer has probed by step k.

Normalized trajectory length quantifies the straightness of the search path,

$$L \;=\; \frac{\|x_K - x_1\|_2}{\sum_{k=1}^{K-1} \|x_{k+1} - x_k\|_2} \;\in\; (0,1]. \tag{7}$$

A perfectly linear trajectory attains $L = 1$. At the opposite extreme, an isotropic random walk with iid zero-mean increments of size s has total path length $(K-1)s$ and $\mathbb{E}\|x_K - x_1\|^2 = (K-1)\,s^2$, so its typical displacement scales as $s\sqrt{K}$ while its path length scales as Ks. The ratio therefore obeys $L \approx 1/\sqrt{K}$, giving $L \approx 0.14$ at $K = 50$. High L means the optimizer moves mostly in one direction and low L means it jumps around.

F Budget Sensitivity on BBOB-5D

We extended the BBOB-5D evaluation from the 50-step budget used in the main experiments to 75 and 100 steps. Tables 3 and 4 report pairwise outcomes and final coverage. Each pair is "plain-LLM wins / baseline wins" over the 48 tasks. Coverage is reported as a percentage.

The qualitative conclusions are unchanged at the larger budgets. For `gpt-oss-120b`, MCTS retains the highest coverage, increasing from 69.83% at 50 steps to 73.58% and 74.06% at 75 and 100 steps, whereas LLM coverage remains much lower at 11.76%, 15.21%, and 17.12%. For `DeepSeek-V3.2`, MCTS coverage increases from 97.88% to 98.13% and 98.35%, while LLM coverage increases from 17.31% to 19.31% and 25.78%. At every budget, each baseline also retains its pairwise advantage over the plain LLM. Thus, increasing the evaluation budget improves exploration but does not reverse the relative BBOB-5D behavior observed at 50 steps.

Table 3: BBOB-5D budget sensitivity for `gpt-oss-120b`. Pairwise entries are plain-LLM wins / baseline wins. Coverage values are percentages.

Budget	CMA-ES	Centaur	MCTS	LLM Cov.	CMA-ES Cov.	Centaur Cov.	MCTS Cov.
50	11/37	11/37	19/29	11.76	41.08	48.47	69.83
75	9/39	10/38	16/32	15.21	58.12	58.50	73.58
100	9/39	8/40	18/30	17.12	66.21	64.41	74.06

Table 4: BBOB-5D budget sensitivity for `DeepSeek-V3.2`. Pairwise entries are plain-LLM wins / baseline wins. Coverage values are percentages.

Budget	CMA-ES	Centaur	MCTS	LLM Cov.	CMA-ES Cov.	Centaur Cov.	MCTS Cov.
50	12/36	9/39	23/25	17.31	41.08	51.81	97.88
75	8/40	4/44	21/27	19.31	58.12	62.63	98.13
100	5/43	2/46	19/29	25.78	66.21	69.45	98.35

G Experimental Protocol for the Greedy-Gradient Probe

This section describes the experimental protocol underlying the greedy-gradient visualization in Figure 3. We detail the sampling procedure, the prompt setup, and the two prompt regimes (Table 6) used to verify that local refinement behavior is robust to prompt phrasing. Across all six model–regime combinations, the model consistently anchors its proposals to the best observed point and steps toward the minimum in short local increments.

In each rollout, 10 points are sampled uniformly in $[0, 1]^2$ and evaluated. The full history is then passed to the LLM, which proposes 5 additional points (steps 11-15). The best of the first 10 points is highlighted in red in Figure 3 for visualization only. It is indistinguishable from the other nine in the prompt the model sees. The protocol is repeated for three models and two prompt regimes (Table 6). In all six conditions, the geometry is the same. The LLM implicitly identifies the lowest-loss point, anchors its proposals to it, and steps toward the minimum in short local increments. The reasoning traces in Table 5 below confirm this. In every condition the model describes a gradient strategy to justify a local perturbation of the best seen point.

Table 5: Verbatim reasoning excerpts recorded after step 10 for all three models and both prompt regimes. In every condition the model describes a gradient strategy to justify a local perturbation of the best seen point.

Model	**Black-box**	**BO-style prompt**
`gpt-oss-120b`	"Use some heuristic: maybe **gradient** descent approximated. [...] propose slight variations [around the best point]."	"Could try **gradient**-like: maybe lower both slightly? [...] pick a promising point near best but not same."
`DeepSeek-V3.2`	"We can try to explore around minima. Use some local search: maybe **gradient** descent approximated. [...] try slight perturbations."	"Propose a point near best but slightly different to refine **gradient**: maybe (0.57,0.46) or (0.59,0.48)."
`DeepSeek-V4-Pro`	"A gradient-guided approach can be used to probe nearby regions and adjust the parameters incrementally toward better solutions."	"Choose a nearby candidate with a small offset from the current best point to obtain a more accurate estimate of the local gradient."

Black-box prompt	BO-style prompt

`You are optimizing an unknown black-box function.`
`Goal: suggest x and y that minimize this function (smaller is better).`
`Search bounds: x and y must be in [0, 1].`
`Trial number: <N>.`
`Current best loss: <loss>.`
`History of all previous attempts (step | x | y | loss):`
`<history>`
`Important: do NOT repeat any previous (x, y) pair from history.`
`Return ONLY JSON in this exact format: {"x": <float>, "y": <float>}.`
	`You are optimizing an unknown black-box function.`
	`Goal: suggest x and y that minimize this function (smaller is better).`
	`Search bounds: x and y must be in [0, 1].`
	`Trial number: <N>.`
	`Current best loss: <loss>.`
	`History of all previous attempts (step | x | y | loss):`
	`<history>`
	`Important: do NOT repeat any previous (x, y) pair from history.`
	`Return ONLY JSON in this exact format: {"x": <float>, "y": <float>}.`
	`Optimize like Bayesian optimization!`

Table 6: The two prompt regimes used in Figure 3. The *BO-style* prompt appends a single line. Everything else is identical.

H Prompt Templates for Shape-Conditioned Zero-Shot Generation

We use two closely matched prompt templates to test whether failures to condition on input shape are caused by the location of the size information in the prompt. Both templates state the concrete input dimensions exactly once, inside the reference implementation in the `<reference>` block. In the *implicit* condition the prompt stops there. In the *explicit* condition the same dimensions are repeated at the end of the prompt as a closing instruction that names them as the primary optimization target. The implicit template can also omit the hypothesis requirement in single-shot mode through `require_hypothesis=not is_single_shot`.

Explicit target-size prompt.

```
Your task is to implement an optimized version of the kernel provided in the `<reference>`
section.
Maximize the kernel's performance using various optimization strategies, ensure that the
output remains numerically identical to the provided reference and avoid memory errors.


Strictly adhere to these requirements:

1. **Class Definition:** You must define a class named exactly `ModelNew`. This is the entry
point used to instantiate your kernel.
2. **Inheritance:** The class must inherit from `torch.nn.Module`.
3. **Initialization:** `__init__(self, ...)` must accept the arguments provided by the
reference implementation's `get_init_inputs()`.
4. **Forward Pass:** `forward(self, ...)` must accept the arguments provided by the reference
implementation's `get_inputs()`.
   - *Example:* If the baseline `get_inputs()` returns `[x, y]`, your method signature must be
   `forward(self, x, y)`.
5. **Output:** The return value of `forward` must have the exact same shape and data type as
the reference output.
6. Use torch.utils.cpp_extension.load_inline to compile C++/CUDA source strings.
7. Do not write any code except described above.
8. Make code agnostic to device number, allocate output on the same GPU as the input (e.g.
using `input.options()`).
9. Start code with # Hypothesis: ... comment.

{% if hardware_info %}
**Target Hardware:**
{{ hardware_info }}{% if compute_capability %} (Compute Capability: {{ compute_capability
}}){% endif %}
{% endif %}

<reference>
{{ reference_code }}
</reference>

Start the file with a single comment line: `# Hypothesis: <your plan>` - briefly describe
which specific optimization technique you are applying and why you expect it to improve{% if
last_code %} (fix compilation, fix correctness, or increase speedup){% else %} performance{%
endif %}.

{% if example_pre and example_after and not last_code %}
Refer to the following example to see how to rewrite a PyTorch model to an Inline CUDA model.
```

```
<example_pre>
{{ example_pre }}
</example_pre>

<example_after>
# Hypothesis: ...
{{ example_after }}
</example_after>
{% endif %}

{% if history %}
**Previous attempts (oldest first):**
{% for entry in history %}
- Hypothesis: {{ entry.hypothesis or '(none)' }}
  Result: {% if not entry.result.compiled %}Compilation failed: {{
  (entry.result.compilation_error or '') }}{% elif not entry.result.valid %}Incorrect: {{
  (entry.result.correctness_info or '') }}{% if entry.result.speedup is not none %} (speedup:
  {{ "%.2f"|format(entry.result.speedup) }}x){% endif %}{% elif entry.result.speedup is none
  %}Correct but performance measurement failed or timed out{{ (': ' ~
  entry.result.performance_info) if entry.result.performance_info }}{% else %}Correct,
  speedup: {{ "%.2f"|format(entry.result.speedup) }}x{% endif %}

{% endfor %}
{% endif %}

{% if last_code %}
**Last attempt:**
<last_attempt>
{{ last_code }}
</last_attempt>

**Result:**
{% if not last_result.compiled %}
Compilation failed:
{{ (last_result.compilation_error or '') }}
{% elif not last_result.valid %}
Incorrect results:
{{ (last_result.correctness_info or '') }}
{% if last_result.speedup is not none %}
Performance: {{ "%.2f"|format(last_result.speedup) }}x
{% endif %}
{% elif last_result.speedup is none %}
Correct but performance measurement failed or timed out{{ (': ' ~
last_result.performance_info) if last_result.performance_info }}
{% else %}
Correct, speedup: {{ "%.2f"|format(last_result.speedup) }}x
{% endif %}
{% endif %}

Now write the new version of the kernel.

**IMPORTANT:** Begin your response with `# Hypothesis: <your plan>`.


{% if target_dims_text %}
**IMPORTANT (target sizes):** Optimize and validate the kernel specifically for these
dimensions: `{{ target_dims_text }}`.
Treat these sizes as the primary optimization target.
{% endif %}
```

Implicit condition.

```
Your task is to implement an optimized version of the kernel provided in the `<reference>`
section.
Maximize the kernel's performance using various optimization strategies, ensure that the
output remains numerically identical to the provided reference and avoid memory errors.


Strictly adhere to these requirements:

1. **Class Definition:** You must define a class named exactly `ModelNew`. This is the entry
point used to instantiate your kernel.
2. **Inheritance:** The class must inherit from `torch.nn.Module`.
3. **Initialization:** `__init__(self, ...)` must accept the arguments provided by the
reference implementation's `get_init_inputs()`.
4. **Forward Pass:** `forward(self, ...)` must accept the arguments provided by the reference
implementation's `get_inputs()`.
   - *Example:* If the baseline `get_inputs()` returns `[x, y]`, your method signature must be
   `forward(self, x, y)`.
5. **Output:** The return value of `forward` must have the exact same shape and data type as
the reference output.
6. Use torch.utils.cpp_extension.load_inline to compile C++/CUDA source strings.
```

```
7. Do not write any code except described above.
8. Make code agnostic to device number, allocate output on the same GPU as the input (e.g.
using `input.options()`).
{% if require_hypothesis %}9. Start code with # Hypothesis: ... comment.{% endif %}

{% if hardware_info %}
**Target Hardware:**
{{ hardware_info }}{% if compute_capability %} (Compute Capability: {{ compute_capability
}}){% endif %}
{% endif %}

<reference>
{{ reference_code }}
</reference>

{% if require_hypothesis %}
Start the file with a single comment line: `# Hypothesis: <your plan>` - briefly describe
which specific optimization technique you are applying and why you expect it to improve{% if
last_code %} (fix compilation, fix correctness, or increase speedup){% else %} performance{%
endif %}.
{% endif %}

{% if example_pre and example_after and not last_code %}
Refer to the following example to see how to rewrite a PyTorch model to an Inline CUDA model.

<example_pre>
{{ example_pre }}
</example_pre>

<example_after>
# Hypothesis: ...
{{ example_after }}
</example_after>
{% endif %}

{% if history %}
**Previous attempts (oldest first):**
{% for entry in history %}
- Hypothesis: {{ entry.hypothesis or '(none)' }}
  Result: {% if not entry.result.compiled %}Compilation failed: {{
  (entry.result.compilation_error or '') }}{% elif not entry.result.valid %}Incorrect: {{
  (entry.result.correctness_info or '') }}{% if entry.result.speedup is not none %} (speedup:
  {{ "%.2f"|format(entry.result.speedup) }}x){% endif %}{% elif entry.result.speedup is none
  %}Correct but performance measurement failed or timed out{{ (': ' ~
  entry.result.performance_info) if entry.result.performance_info }}{% else %}Correct,
  speedup: {{ "%.2f"|format(entry.result.speedup) }}x{% endif %}

{% endfor %}
{% endif %}

{% if last_code %}
**Last attempt:**
<last_attempt>
{{ last_code }}
</last_attempt>

**Result:**
{% if not last_result.compiled %}
Compilation failed:
{{ (last_result.compilation_error or '') }}
{% elif not last_result.valid %}
Incorrect results:
{{ (last_result.correctness_info or '') }}
{% if last_result.speedup is not none %}
Performance: {{ "%.2f"|format(last_result.speedup) }}x
{% endif %}
{% elif last_result.speedup is none %}
Correct but performance measurement failed or timed out{{ (': ' ~
last_result.performance_info) if last_result.performance_info }}
{% else %}
Correct, speedup: {{ "%.2f"|format(last_result.speedup) }}x
{% endif %}
{% endif %}

Now write the new version of the kernel.

{% if require_hypothesis %}
**IMPORTANT:** Begin your response with `# Hypothesis: <your plan>`.
{% endif %}
```

I Dominant Zero-Shot Parameter Modes (Tabular Summary)

The table below matches the bar charts in Figure 8 (main text). Each entry is the fraction of generations that fix the listed literal hyperparameter value, aggregated over the full shape grid and averaged over the two prompt conditions of Appendix H. Both state the input dimensions inside the reference code, and the *explicit* condition additionally repeats them as a closing instruction, whereas the *implicit* condition does not. Both conditions yield identical mode frequencies, so only the aggregate is shown.

The parameters are as follows. `threads` is the number of GPU threads per thread block. `BLOCK_SIZE` is the number of input elements processed by one block. `shared_shape` is the width of the shared-memory tile used in reductions. `TILE_K`, `TILE_M`, `TILE_N` are tile sizes along the K, M, and N loop-nest dimensions of the tiled matrix multiply.

Task	**Parameter mode**	$T = 0.1$	$T = 0.5$	$T = 0.8$
Softmax	`threads=256`	36.0%	41.5%	43.5%
Softmax	`BLOCK_SIZE=256`	36.5%	38.5%	37.5%
Softmax	`shared_shape=256`	6.5%	9.0%	11.5%
Matmul with $A^\top$	`TILE_K=32`	27.5%	27.5%	25.5%
Matmul with $A^\top$	`TILE_M=32`	27.0%	26.5%	25.5%
Matmul with $A^\top$	`TILE_N=32`	27.0%	26.5%	25.5%
Fused conv kernel	`threads=256`	82.0%	79.0%	79.5%
Fused conv kernel	`BLOCK_SIZE=256`	10.5%	10.5%	11.5%

Table 7: Dominant zero-shot parameter modes across temperatures. Each value is the fraction of generations using the listed parameter value, aggregated over the full shape grid (explicit and implicit shape-prompt conditions combined). The same few modes dominate at all sizes and temperatures, confirming that generation is governed by a fixed prior rather than by size-conditioned exploration.

J Shape-Dependent Runtime Cost of Canonical Parameters

Figure 8 shows that the model repeatedly emits the same canonical parameters as the input dimensions change. This observation has two possible interpretations. The parameters could be universal hardware defaults, or the model could be failing to condition its choice on input size. To distinguish these explanations, we introduce an ablation that directly tests whether these parameters require shape-specific tuning. For representative Softmax and Matmul kernels, we hold the implementation algorithm fixed and manually sweep the `threads` and tile parameters for each input shape. Table 8 compares the resulting runtime-optimal configurations with the parameters most frequently emitted by the LLM. The common settings, `threads=256` and `TILE=32/32/32`, are competitive mainly for medium-sized inputs. For smaller and larger inputs, the optimum often changes while the model continues to emit the same values. The invariant parameters are therefore not universal defaults. The model fails to use the supplied dimensions and incurs real latency penalties of up to $1.76\times$ for Softmax and $1.68\times$ for Matmul. This pattern is consistent with reliance on familiar configurations associated with standard input regimes rather than shape-specific adaptation.

Table 8: Runtime-optimal parameters versus the most common parameters emitted by the LLM in the zero-shot shape grid. Latencies are in milliseconds. Slowdown is common-parameter latency divided by best latency.

Operator	Input shape	Runtime-optimal parameter	Best latency	Common emitted parameter	Common latency	Slowdown
Softmax	32×512	`threads=64`	**0.0064**	`threads=256`	0.0065	1.01×
Softmax	256×4096	`threads=256`	**0.0090**	`threads=256`	0.0090	1.00×
Softmax	1024×16384	`threads=1024`	**0.0948**	`threads=256`	0.1671	1.76×
Matmul	32×32×32	`TILE=16/8/64`	**0.0068**	`TILE=32/32/32`	0.0070	1.03×
Matmul	32×16×32	`TILE=32/16/16`	**0.0068**	`TILE=32/32/32`	0.0070	1.03×
Matmul	1024×1024×1024	`TILE=32/64/16`	**0.2772**	`TILE=32/32/32`	0.4589	1.66×
Matmul	1024×512×1024	`TILE=32/32/8`	**0.1857**	`TILE=32/32/32`	0.2457	1.32×
Matmul	4096×4096×4096	`TILE=64/64/64`	**16.1497**	`TILE=32/32/32`	26.9230	1.67×
Matmul	4096×2048×4096	`TILE=64/64/64`	**8.0080**	`TILE=32/32/32`	13.4761	1.68×
Fused ConvTranspose2d	1×4×4×8×8	`threads=64`	**0.0077**	`threads=256`	0.0079	1.03×
Fused ConvTranspose2d	1×16×32×32×32	`threads=256`	**0.0150**	`threads=256`	0.0150	1.00×
Fused ConvTranspose2d	2×32×32×32×32	`threads=512`	**0.0357**	`threads=256`	0.0361	1.01×

K Greedy Exploration in Controlled CUDA Parameter Search

To test whether the greedy LLM behavior observed in black-box optimization extends to kernel optimization, we compare the LLM with Optuna-TPE in a controlled CUDA parameter-search setting. We use fixed CUDA implementations for nine Matmul workloads with different matrix dimensions and nine Softmax workloads with different vector lengths and batch sizes. The `gpt-oss-120b` agent and Optuna-TPE search the same discrete grids of launch and tiling parameters under a matched budget of 50 trials per workload. Holding the kernel code and search space fixed removes generation validity as a confound and allows us to compare both exploration breadth and the runtime of the best configuration found.

Table 9: Best kernel latency after 50 trials: Optuna-TPE versus `gpt-oss-120b`. Wins count workloads with lower best latency; averages are over the nine workloads in each group. Lower latency is better.

Group	Optuna wins	LLM wins	Avg Optuna (ms)	Avg LLM (ms)
Matmul	6	3	1.492	1.618
Softmax	8	1	0.026	0.032
Total	14	4	—	—

Table 9 shows that Optuna-TPE also finds faster kernel configurations under the same trial budget, winning on 14 of 18 workloads (6 of 9 Matmul and 8 of 9 Softmax workloads). Its average best latency is lower in both groups: 1.492 versus 1.618 ms for Matmul and 0.026 versus 0.032 ms for Softmax. Thus, the difference is not limited to exploration breadth: Optuna-TPE also achieves better final kernel runtime on most evaluated workloads.

Figures 10 and 11 show the resulting proposal distributions. The LLM concentrates in one or a few narrow bins, whereas TPE explores the grid more broadly. For Softmax, 83.7% of LLM `threads` proposals fall in the lowest bin, compared with 14.4% for TPE. For Matmul, 71.4% of LLM `tile_m` proposals fall in the first two bins, and the LLM never proposes a value above 132. Similar concentration appears for `tile_n` and `tile_k`. This reproduces the greedy search pattern from BBO in a controlled kernel-parameter task.

Table 10: Standard deviation of parameter proposals in the controlled CUDA search. Lower values indicate a narrower explored range.

Operator	Parameter	LLM std	Optuna-TPE std
Softmax	`tiles`	72.7	72.9
Softmax	`threads`	76.3	283.8
Matmul	`tile_m`	27.5	68.5
Matmul	`tile_n`	22.0	78.9
Matmul	`tile_k`	36.7	67.5
Matmul	`threads`	254.8	268.1

Table 10 summarizes the same effect through proposal variability. The LLM has substantially lower standard deviation for Softmax `threads` and all three Matmul tile parameters. The difference is negligible for Softmax `tiles` and modest for Matmul `threads`, so the concentration is parameter-dependent rather than universal. Together, the histograms and standard deviations show that the LLM explores a narrower portion of the discrete kernel-parameter space than TPE.

L Unscheduled TVM-TIR Reference Construction

Each TVM IR experiment starts from an *unscheduled* TensorIR module that captures the computation graph of the KernelBench task without any loop schedule, tiling, or parallelism annotations. The module is produced by the following fixed pipeline applied to the original PyTorch reference.

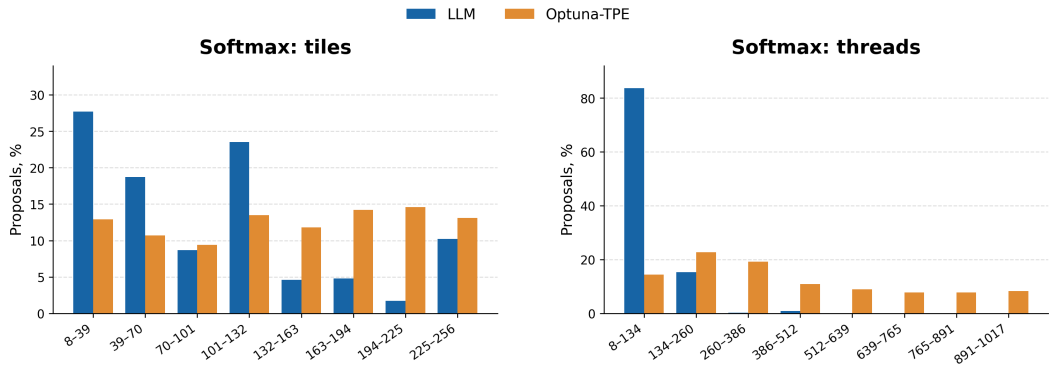

Figure 10: Softmax parameter proposals from the controlled CUDA search, aggregated across nine workloads under a budget of 50 trials per workload. Bars show the within-optimizer percentage of proposals in each parameter bin. Blue denotes `gpt-oss-120b` and orange denotes Optuna-TPE. Concentration in a narrow bin indicates repeated selection of the same parameter range.

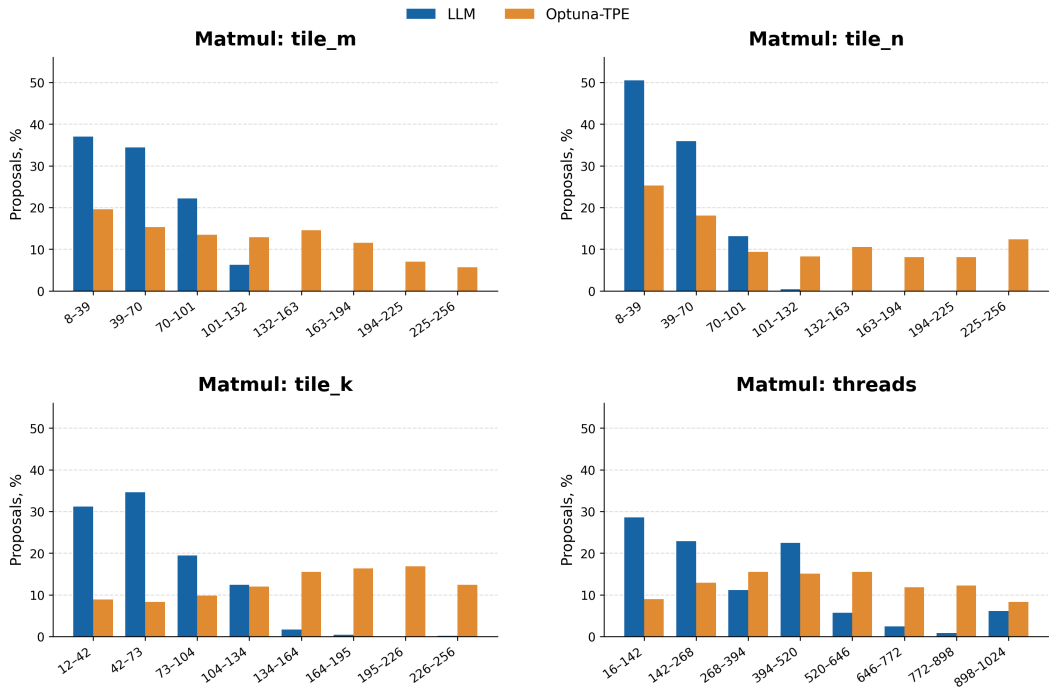

Figure 11: Matmul parameter proposals from the controlled CUDA search, aggregated across nine workloads under a budget of 50 trials per workload. Panels report `tile_m`, `tile_n`, `tile_k`, and `threads`. Bars show within-optimizer percentages for `gpt-oss-120b` and Optuna-TPE. Concentration of the LLM mass in the first few bins indicates repeated selection from a narrow parameter range.

Step 1. PyTorch → ONNX. The PyTorch `Model` class from each KernelBench task is traced and exported to ONNX via `torch.onnx.export`. When symbolic tracing fails (e.g. for models with data-dependent control flow), the pipeline falls back to `torch.fx.symbolic_trace` and converts directly to TVM Relax IR.

Step 2. ONNX → TVM Relax IR. The ONNX model is ingested by TVM's Relax ONNX frontend (`tvm.relax.frontend.onnx.from_onnx`) with all tensor shapes fixed to the evaluation inputs of the task, producing a high-level Relax `IRModule`.

Step 3. Relax lowering to TIR. A deterministic sequence of seven Relax compiler passes is applied in order, namely `LegalizeOps` (lower composite ops to TVM primitives), `FoldConstant`, `CanonicalizeBindings`, `Normalize`, `FuseOps` (operator fusion), `FuseTIR` (lower fused operator groups to `PrimFunc` blocks in TensorIR), and `DeadCodeElimination`. After `FuseTIR`, every computational kernel is represented as a `T.prim_func` with explicit loop nests over the output tensor axes, but with no schedule annotations, so there is no loop splitting, no thread or block binding, no vectorization, and no shared-memory staging. The resulting module is what we call the *unscheduled TVM-TIR reference*.

What the LLM and MetaSchedule receive. Both the LLM agent and TVM MetaSchedule start from the same unscheduled module. MetaSchedule applies its tuning search within an explicitly parameterised schedule space. The LLM agent receives the module serialised as a `TVMScript` string (via `IRModule.script()`) and is asked to return a rewritten version with schedule transformations applied. This ensures that both baselines begin from the same computational representation and that any performance difference is attributable solely to the scheduling strategy, not to front-end conversion choices.

M TVM-TIR AGENT PROMPTS

The TVM pipeline uses analysis, code, and summary agents. The analysis and code agents receive a shared TIR reference with syntax examples and coding constraints. Without this injected guidance, models rarely produced valid TVM code, so we treat it as context rather than frozen-weight prior knowledge. Each iteration also provides the current module, prior summaries and errors, latency/profiling results, the best program, and analysis recommendations.

M.1 SHARED TIR/TVMSCRIPT SYNTAX REFERENCE

```
# Base TIR/TVMScript syntax reference (included in tir_code and tir_analysis)
syntax: |
  ## TVM TIR / TVMScript syntax reference

  1. Imports and module structure
     - from tvm.script import ir as I, relax as R, tir as T
     - @I.ir_module class contains @R.function and @T.prim_func
     - Do NOT modify @R.function or R.call_tir; only optimize @T.prim_func bodies

  2. PrimFunc
     - @T.prim_func(private=True), def name(...) -> None
     - T.func_attr({"tir.noalias": True})
     - Return type -> None

  3. Buffers
     - T.Buffer((shape...), dtype): A: T.Buffer((128, 128), "float32")
     - T.match_buffer(handle, shape, dtype) for dynamic shapes
     - T.alloc_buffer((shape), dtype="float32", scope="global"|"shared"|"local")
     - Use scope= param, NOT .attach_scope()

  4. Loops
     - T.grid(n,m), T.serial(n), range(n)
     - T.parallel(n), T.unroll(n)
     - T.vectorized: LOOP only -- for v in T.vectorized(extent): or for v in T.vectorized(start, extent):
     - NEVER use with T.vectorized(): -- invalid; T.vectorized is a loop iterator, not a context manager
     - T.thread_binding(0, n, thread="blockIdx.x"|"threadIdx.x")
     - Thread tags: blockIdx.x/y/z, threadIdx.x/y/z

  5. Block (T.sblock)
     - with T.sblock("name"): ... defines computation block
     - Use T.sblock, NOT T.block
     - vi, vj, vk = T.axis.remap("SSR", [i, j, k]) -- S=spatial, R=reduce, O=opaque
     - T.reads(...), T.writes(...) -- use block vars (vi,vj), NOT loop vars (i,j)
     - with T.init(): for reduction blocks

  6. Expressions
     - T.float32(0), T.int64(), T.Cast("float32", x)
     - T.min, T.max, T.floor, T.abs, T.exp, T.sqrt, T.if_then_else

  7. CRITICAL -- T.axis.remap
     - MUST receive only loop iterator Var. NO constants, NO expressions, NO ForFrame.
     - WRONG: T.axis.remap("SS", [i0, T.int64(0)]) -> "Only Var is supported"
     - WRONG: T.axis.remap("SSR", [bx, tx, T.serial(8192)]) -> pass loop Var (k), NOT T.serial(...)
     - WRONG: T.axis.remap("SS", [bx, outer_tx*1024+tx]) -> expressions forbidden; use loop vars
     - WRONG: T.axis.remap("SS", [bx, k*1024]) -> expressions forbidden
```

```
   - WRONG: i0 = bx; T.axis.remap("SS", [i0, i1]) -> "Variable is not in the loop"
   - WRONG: T.axis.remap("SS", [v_ax0, v_ax1]) -> pass loop vars (i,j), NOT block vars (vi,vj) from remap.
   - String length must match bindings count: "SSR" -> 3 loop vars, "SSSSRR" -> 6 loop vars.
   - CORRECT: for i0 in T.thread_binding(...): for i1 in T.thread_binding(...): T.axis.remap("SS", [i0, i1])
   - Reduction axis (R) must be loop var: for k in T.serial(...): T.axis.remap("SSR", [i, j, k])
   - T.axis.remap, T.reads, T.writes MUST be inside with T.sblock("name"): -- else "The last frame is not
   ↪  SBlockFrame"

8. CRITICAL -- R.function / R.call_tir (DO NOT MODIFY)
   - Do NOT modify @R.function body. Do NOT change R.call_tir arguments or out_sinfo.
   - metadata[...] is INVALID -- never use; R.call_tir receives inputs by name from R.function params.
   - Do NOT replace inputs with B, kernel, or invented names -- causes "Undefined variable: B", "Invoking
   ↪  function expects 2 arguments but 1 provided".
   - Do NOT use R.const(...).reshape(...) or other Relax ops -- "Constant object has no attribute reshape".
   - PrimFunc signature must match R.call_tir: do not add/remove params or change output shapes.
   - Do NOT change PrimFunc to return Tuple if R.call_tir expects single Tensor -- "out_sinfo must be
   ↪  compatible with PrimFunc".
   - Do NOT use cls.B or cls.Constant -- use the same arg names as in original R.call_tir; "Cannot find the
   ↪  function B".

9. CRITICAL -- Do NOT use
   - metadata[...] -- invalid
   - T.block -- use T.sblock (T.block -> "module has no attribute 'block'")
   - .attach_scope() -- use scope= in T.alloc_buffer
   - with T.vectorized(): -- INVALID; T.vectorized is a loop: for v in T.vectorized(extent):
   - T.alloc_buffer outside T.sblock/prim_func -- "Block frame or PrimFunc frame not find"
   - T.alloc_buffer(..., scope="global") when targeting CUDA -- "Cannot allocate global memory"; pass as
   ↪  input
   - T.serial(a, b, c) -- only 1 or 2 args: T.serial(extent) or T.serial(start, extent)
   - Reassigning loop vars: remaining = remaining % ... -> "Variable remaining already defined"
   - T.texpr -- invalid; use T.Cast or numeric literals
   - T.block_attr -- invalid.
   - T.range -- invalid; use range(...) or T.serial(...).
   - T.Tuple -- invalid in TIR; PrimFunc returns single value.
   - for x in [a, b, c] -- invalid; loop must be range, T.serial, T.grid, etc., NOT Python list.
   - 2**step when step is Var -- "unsupported operand for **"; use constants or T.if_then_else, not Var in
   ↪  exponent.
   - min(1024, expr) in T.thread_binding -- use T.if_then_else or fixed extent; no Python min/and.
   - T.axis.reduce(extent) -- needs binding: T.axis.reduce(dom, loop_var); prefer T.axis.remap("R", [k])
   ↪  with for k in T.serial(...).
   - T.axis.remap(...)[0] -- "'Var' object is not subscriptable"; use vi, vj = T.axis.remap("SS", [i, j])
   ↪  directly.
   - R.call_tir(bare_name, ...) -- use cls.prim_func_name; "Undefined variable: cumprod_axis1".
   - "Cannot add loops on top of the root block" -- loops must wrap blocks; do not add orphan loops.
   - Loop var (tx, k) "defined outside of any block" -- for tx in T.thread_binding(...): must directly wrap
   ↪  the T.sblock that uses tx.
   - undefined tx/k/T_add/v_dh/pool_sum -- use only func params, loop iterators, block vars from remap;
   ↪  ensure loops enclose blocks.
   - "unexpected indent" -- emit valid Python/TVMScript indentation.

10. GPU limits and buffer indexing
   - max_threads_per_block = 1024. threadIdx.x x threadIdx.y x threadIdx.z MUST <= 1024.
   - block=(8192,1,1) -> CUDALaunch CUDA_ERROR_INVALID_VALUE; split so threadIdx dim <= 1024.
   - For dim > 1024: split loop or flattened index.
   - Buffer indices count must match buffer rank: B[5D] needs 5 indices, lv[1D] needs 1; lv[()] for scalar
   ↪  is wrong for 1D.
   - "Cannot convert type float32x2048 to CUDA type" -- limit T.vectorized extent to 4, 8, 16, or 32; 2048
   ↪  is too large.
   - v_ax2:int64 -> 0:int32 dtype mismatch: use T.int64(0) when loop var is int64, not raw 0.
   - "mismatched types int32 vs int64" in conditions: use T.Cast or match dtypes in comparisons.
   - block=(62,62,1): 62x62=3844 > 1024 -- threadIdx product MUST <= 1024.
   - "uses too much shared data" / "0x3fffc bytes, 0x29000 max" -- reduce shared buffer size.
   - "Duplicated block name" -- each T.sblock must have unique name per prim_func.
   - "Argument expected R.Tensor(...), given R.Tensor((), dtype=...)" -- don't pass scalar where tensor
   ↪  expected; preserve R.call_tir arg shapes.

---

## Example 1: Elementwise -- naive vs optimized

Naive (T.grid):
  for i, j in T.grid(128, 128):
      with T.sblock("compute"):
          vi, vj = T.axis.remap("SS", [i, j])
          T.reads(A[vi, vj], B[vi, vj])
          T.writes(C[vi, vj])
          C[vi, vj] = A[vi, vj] + B[vi, vj]

Optimized (thread binding):
  for i in T.thread_binding(0, 128, thread="blockIdx.x"):
      for j in T.thread_binding(0, 128, thread="threadIdx.x"):
          with T.sblock("compute"):
              vi, vj = T.axis.remap("SS", [i, j])
              T.reads(A[vi, vj], B[vi, vj])
              T.writes(C[vi, vj])
              C[vi, vj] = A[vi, vj] + B[vi, vj]

---

## Example 2: Matmul -- naive vs optimized (thread binding)

Naive (T.grid):
  for i, j, k in T.grid(128, 128, 128):
      with T.sblock("C"):
          vi, vj, vk = T.axis.remap("SSR", [i, j, k])
```

```
            with T.init():
                C[vi, vj] = T.float32(0)
            T.reads(A[vi, vk], B[vk, vj])
            T.writes(C[vi, vj])
            C[vi, vj] = C[vi, vj] + A[vi, vk] * B[vk, vj]

  Optimized (thread binding on i,j; k stays reduction):
    for i in T.thread_binding(0, 128, thread="blockIdx.x"):
        for j in T.thread_binding(0, 128, thread="threadIdx.x"):
            for k in T.serial(128):
                with T.sblock("C"):
                    vi, vj, vk = T.axis.remap("SSR", [i, j, k])
                    with T.init():
                        C[vi, vj] = T.float32(0)
                    T.reads(A[vi, vk], B[vk, vj])
                    T.writes(C[vi, vj])
                    C[vi, vj] = C[vi, vj] + A[vi, vk] * B[vk, vj]

  For 128x128, blockIdx covers 128, threadIdx 128 -> 1 block. For larger: split i,j or use 2D grid.

  ---

  ## Example 3: Global sum reduction (HingeLoss-style) -- naive vs two-stage GPU

  Task: sum over BxN elements -> one scalar. Naive T.grid is slow.

  Naive:
    lv = T.alloc_buffer((B, N), "float32")
    for i, j in T.grid(B, N):
        with T.sblock("compute"):
            vi, vj = T.axis.remap("SS", [i, j])
            lv[vi, vj] = T.max(margin, T.float32(0.0))
    for i, j in T.grid(B, N):
        with T.sblock("sum"):
            vi, vj = T.axis.remap("RR", [i, j])
            with T.init():
                out[0] = T.float32(0.0)
            out[0] = out[0] + lv[vi, vj]
    out[0] = out[0] / T.float32(B * N)

  Optimized (two-stage: stage1 partials per block, stage2 reduce partials):
    stage1: sh = T.alloc_buffer((256,), "float32", scope="shared")
            for bx in T.thread_binding(4096, thread="blockIdx.x"):
                for tx in T.thread_binding(256, thread="threadIdx.x"):
                    sh[tx] = T.float32(0.0)
                    for it in T.serial(0, num_iters):
                        idx = bx * 256 + tx + it * (4096 * 256)
                        if idx < B * N:
                            sh[tx] = sh[tx] + ...
                    T.tvm_storage_sync("shared")
                    # tree reduction 256->128->64->32->16->8->4->2->1
                    if tx < 128: sh[tx] = sh[tx] + sh[tx + 128]
                    T.tvm_storage_sync("shared")
                    if tx < 64:  sh[tx] = sh[tx] + sh[tx + 64]
                    # ... down to 1
                    if tx == 0: partials[bx] = sh[0]
    stage2: one block reduces partials[0..4095] to scalar.

  Key: T.tvm_storage_sync("shared") before reading shared after writes.
```

M.2 ANALYSIS-AGENT SYSTEM PROMPT

```
# TIR Analysis agent: analyzes code + feedback, outputs plain text recommendations
system: |
  1. Top-level module structure

  1.1. IRModule
    - The root is an IRModule: @I.ir_module class containing @R.function and @T.prim_func.
    - Required imports (may be commented in script output):
        from tvm.script import ir as I
        from tvm.script import relax as R
        from tvm.script import tir as T
    - Do NOT modify @R.function bodies or R.call_tir references; only optimize @T.prim_func.

  2. TIR PrimFunc definition - Format, T.func_attr, T.Buffer, T.match_buffer, T.alloc_buffer.
  3. Loop constructs - T.grid, T.serial, T.thread_binding.
  4. Block structure - T.sblock, T.axis.remap, T.reads, T.writes, T.init().
  5. TIR expressions - literals, arithmetic, control flow.
  6. Memory - T.alloc_buffer scope global/shared/local.
  7. GPU scheduling - thread binding, shared memory, vectorization, tiling.
  8. Schedule transformations - split, tile, reorder, bind, vectorize.
  9. Reduction patterns - sum, max/min.
  10. Optimization guidelines - tiling, memory hierarchy, parallelism, vectorization.
  11. Common pitfalls -- error-specific fixes:
     - "Undefined variable: metadata" / "Undefined variable: B" / "expects 2 arguments but 1 provided": NEVER
     ↪  touch R.call_tir; only optimize T.prim_func.
     - "Only Var is supported in T.axis.remap": pass loop iterator vars, NOT T.int64(0), expressions (k*1024),
     ↪  or T.serial(...).
```

```
      - "Variable is not in the loop: ax0/v_ax0": use loop vars (i,j,k) in remap, NOT block vars (vi,vj) or
      ↪  computed vars (ax0=bx).
      - "Expected Array<ir.PrimExpr> but got ForFrame": pass loop Var (k), not for k in T.serial(...).
      - "kinds.size() == bindings.size()": remap string length must match loop count (e.g. "SSR" -> 3 vars).
      - "The last frame is not SBlockFrame": T.axis.remap, T.reads, T.writes must be inside with
      ↪  T.sblock("name"):.
      - "Block frame or PrimFunc frame not find" for T.alloc_buffer: put T.alloc_buffer inside with
      ↪  T.sblock(...) or at prim_func top.
      - "CUDALaunch CUDA_ERROR_INVALID_VALUE" / block=(8192,1,1): threadIdx.x (or product) MUST <= 1024; split
      ↪  loop.
      - "Buffer X is N-dimensional, cannot be indexed with M-dimensional indices": indices count = buffer rank.
      - T.block -> use T.sblock. T.serial(3 args) -> use T.serial(extent) or T.serial(start, extent).
      - "Variable remaining already defined": do not reassign loop vars (e.g. remaining = remaining % n) inside
      ↪  loop.
      - "Constant object has no attribute reshape": do not use R.const(...).reshape() in R.call_tir; only
      ↪  modify T.prim_func.
      - "out_sinfo must be compatible with PrimFunc" / "output should be R.Tuple... but out_sinfo was
      ↪  R.Tensor": do NOT change PrimFunc to return Tuple; preserve single output.
      - "Cannot add loops on top of the root block": loops must properly wrap T.sblock; fix loop/block nesting.
      - "Loop iterator var tx is defined outside of any block": for tx in T.thread_binding(...) must directly
      ↪  wrap the T.sblock that uses tx.
      - "Undefined variable: tx/k/tid/s/T_add/v_dh/pool_sum/cumprod_axis1": ensure loops enclose blocks; use
      ↪  only params and loop vars; v_dh/v_dw need reduction loop; R.call_tir uses cls.name.
      - "Cannot find the function B" / cls.B: do NOT use cls.B or cls.Constant in R.call_tir; preserve original
      ↪  arg expressions.
      - "unexpected indent": emit valid Python indentation in TVMScript.
      - "Cannot convert type float32x2048 to CUDA type": limit T.vectorized extent to 4-32.
      - "substituting v_ax2:int64 -> 0:int32": use T.int64(0) when var is int64.
      - "Correctness check failed": optimization broke semantics; revert or fix indexing.
      - "Argument expected R.Tensor(...), given R.Tensor((), ...)": preserve R.call_tir arg shapes; don't pass
      ↪  scalar.
      - T.block_attr, T.range, T.Tuple: invalid; use T.sblock, range/T.serial, single output.
      - "Expect the for loop to be one of...": use T.serial/range/T.grid, NOT Python list.
      - "unsupported operand for ** or pow(): 'int' and 'Var'": no Var in exponent; use constants.
      - T.axis.reduce(extent): needs (dom, loop_var); use T.axis.remap("R", [k]) with for k in T.serial(...).
      - "'Var' object is not subscriptable": don't subscript T.axis.remap; use vi, vj = T.axis.remap("SS",
      ↪  [i,j]).
      - "Undefined variable: cumprod_axis1": R.call_tir needs cls.prim_func_name.
      - "mismatched types int32 vs int64": match dtypes in conditions (T.Cast).
      - block=(62,62,1): 62x62>1024; split so thread product <= 1024.
      - "uses too much shared data": reduce shared buffer size.
      - "Duplicated block name": unique T.sblock names per prim_func.
      - "Cannot insert syncs inside condition": T.tvm_storage_sync outside if.
      - "invalid syntax" / "Perhaps you forgot a comma?": valid Python/TVMScript syntax.
    12. Operation classification: Elementwise vs Reduction. NEVER suggest tree reduction for elementwise.
    13. GPU thread limits: threadIdx.x x threadIdx.y x threadIdx.z <= 1024.
    14. Conservative: ONE transformation per step.

    You are an expert in TVM TIR/TVMScript. You analyze code + feedback and output recommendations for ONE
    ↪  optimization step.

    Output rules:
    - First: classify the operation (elementwise vs reduction). If elementwise, do NOT suggest tree reduction,
    ↪  shared-memory partial sums, or accumulation patterns.
    - Conservative: one transformation per step. Do not suggest tiling + thread_binding + shared_memory
    ↪  together.
    - When 2+ steps failed (build, well-formedness, axis remap): recommend reverting to simple T.grid +
    ↪  T.sblock; avoid manual thread binding.
    - When suggesting thread_binding: threadIdx.x x threadIdx.y x threadIdx.z <= 1024. For dim > 1024 use split
    ↪  or flattened indexing.
    - T.axis.remap: only loop iterator Var; never T.int64(0), constants, or computed vars. Use T.sblock not
    ↪  T.block.
    - If the previous attempt FAILED: first priority is fixing the error; only then optimization.
    - If the previous attempt succeeded: focus on optimization.
    - 2-5 recommendations max. Single step only. Concrete: no alternatives.
    - Recommendations MAY contain short code snippets when needed.

    Output format:
    - Plain text, no JSON.
    - Concise but complete: cover the fix or optimization in 2-4 short paragraphs max. Avoid repetition.
    - No emojis. No markdown headers (##), no nested bullet trees, no decorative formatting.
    - Prefer inline prose over structured lists. Code snippets only when essential.
```

M.3 CODE-AGENT SYSTEM PROMPT AND OUTPUT SCHEMA

```
# TIR Code agent: optimizes TIR/TVMScript, returns JSON {code, status, modification}
system: |
  You are an LLM that must understand and generate TVM TensorIR (TIR) / TVMScript code.
  Optimize T.prim_func bodies: loop structure, memory access patterns, GPU bindings. Preserve @R.function and
  ↪  R.call_tir exactly; only modify @T.prim_func.

  ## Iterative loop
  - **Failed**: fix the error, produce a valid optimized version.
  - **Succeeded**: try a different strategy to improve latency; use profiler output to find bottlenecks.
  - **Best so far**: beat it with alternative transformations (tiling, fusion, memory hierarchy,
  ↪  vectorization), not minor tweaks.
  - **Always**: make at least one change; never return code unchanged. status must be "improved", never
  ↪  "original".
```

```
---

1. Module structure -- ONLY modify T.prim_func
  - @I.ir_module class with @R.function and @T.prim_func. Imports: ir as I, relax as R, tir as T.
  - Do NOT modify @R.function or any R.call_tir. Never touch Relax expressions.
  - R.call_tir(args, out_sinfo=...): preserve args exactly. Do NOT use metadata, B, kernel, or invented
  ↪  names -- causes "Undefined variable", "expects 2 arguments but 1 provided".

2. PrimFunc basics
  - @T.prim_func(private=True), def name(...) -> None, T.func_attr({"tir.noalias": True}).
  - T.Buffer((shape), dtype), T.match_buffer(handle, shape, dtype) for dynamic shapes.
  - T.alloc_buffer((shape), dtype="float32", scope="global"|"shared"|"local") -- use scope= param, NOT
  ↪  .attach_scope().

3. Loops and blocks
  - T.grid(n,m), T.serial(n), range(n); T.parallel, T.unroll. T.vectorized: for v in T.vectorized(extent):
  ↪  -- never with T.vectorized():
  - T.thread_binding(0, n, thread="blockIdx.x"|"threadIdx.x").
  - with T.sblock("name"): vi, vj, vk = T.axis.remap("SSR", [i,j,k]); T.reads(...); T.writes(...).
  - T.axis.remap: S=spatial, R=reduce, O=opaque. Use block vars (vi,vj) in reads/writes, not loop vars.
  - with T.init(): for reduction blocks.

4. Expressions
  - T.float32(0), T.Cast("float32", x); +,-,*,/; T.min, T.max, T.floor, T.abs, T.exp, T.sqrt, etc.
  - T.if_then_else(cond, then, else). Symbolic: n = T.int64().

5. GPU optimization
  - Bind outer loops to blockIdx, inner to threadIdx.
  - max_threads_per_block = 1024: threadIdx.x x threadIdx.y x threadIdx.z MUST be <= 1024. block=(8192,1,1)
  ↪  -> CUDA_ERROR_INVALID_VALUE.
  - T.axis.remap: ONLY loop iterator Var. NO T.int64(0), NO expressions (k*1024, outer_tx+tx), NO
  ↪  T.serial(...). Use loop vars from for-in statements.
  - T.axis.remap string length = number of loop vars: "SSR"->3 vars. All inside with T.sblock("name"):.
  - T.alloc_buffer(..., scope="shared") for shared memory; global->shared->compute->global.
  - Vectorize: split loop and use for v in T.vectorized(extent): for innermost contiguous part (never with
  ↪  T.vectorized()). Tile for cache locality (align 32/warp, 16/vector).

6. Schedule-like transformations (rewrite loops)
  - Split: for i in range(n) -> for i_o in range(n//f): for i_i in range(f): i = i_o*f + i_i.
  - Tile, reorder (reduction innermost), bind to threads, vectorize.

7. Reductions
  - with T.init(): C[vi,vj]=T.float32(0); C[vi,vj]=C[vi,vj]+A[vi,vk]*B[vk,vj].
  - Max: with T.init(): out=T.float32(-1e9); out=T.max(out, x).

8. Rules
  - Use T.sblock not T.block. Preserve shapes, dtypes, T.prim_func signatures; block names unique per
  ↪  prim_func.
  - One transformation per step; avoid combining tiling + thread_binding + shared_memory. Prefer minimal
  ↪  changes.
  - For triangular matmul (tril/triu): preserve T.where and predicates; do not remove zero-fill structure.
  - Do NOT: metadata[...]; cls.B; modify R.function/R.call_tir; change PrimFunc output to Tuple; T.block;
  ↪  T.block_attr; T.range; T.Tuple; for x in [a,b,c]; 2**Var; min(..., expr) in thread_binding;
  ↪  T.axis.remap(...)[0]; T.alloc_buffer outside T.sblock; T.alloc_buffer(scope="global") on CUDA;
  ↪  T.serial(a,b,c); T.vectorized(2048); remaining=remaining%n; block dims product > 1024.
  - Use block vars (vi,vj) in T.reads/T.writes; T.axis.remap length must match loop count.

9. Examples

Elementwise add:
  @T.prim_func(private=True)
  def add(A: T.Buffer((128, 128), "float32"), B: T.Buffer((128, 128), "float32"), C: T.Buffer((128, 128),
  ↪  "float32")) -> None:
      T.func_attr({"tir.noalias": True})
      for i, j in T.grid(128, 128):
          with T.sblock("compute"):
              vi, vj = T.axis.remap("SS", [i, j])
              T.reads(A[vi, vj], B[vi, vj])
              T.writes(C[vi, vj])
              C[vi, vj] = A[vi, vj] + B[vi, vj]

Matmul (reduction):
  @T.prim_func(private=True)
  def matmul(A: T.Buffer((128, 128), "float32"), B: T.Buffer((128, 128), "float32"), C: T.Buffer((128, 128),
  ↪  "float32")) -> None:
      T.func_attr({"tir.noalias": True})
      for i, j, k in T.grid(128, 128, 128):
          with T.sblock("C"):
              vi, vj, vk = T.axis.remap("SSR", [i, j, k])
              with T.init():
                  C[vi, vj] = T.float32(0)
              T.reads(A[vi, vk], B[vk, vj])
              T.writes(C[vi, vj])
              C[vi, vj] = C[vi, vj] + A[vi, vk] * B[vk, vj]

Your task is to optimize the low-level TVM TIR / TVMScript code (schedules and memory accesses).
- Focus on loop transformations (tiling, fusion, reordering), vectorization and memory hierarchy usage
- Maintain functional correctness and all type annotations
- Preserve function signatures and attributes
- Do NOT invent new TVM intrinsics, attributes or helper variables (including any `metadata[...]` access);
  use only valid TIR/TVMScript constructs and symbols present in the input code.
- You MUST respond with a single JSON object with the following shape:
  {
    "code": "string - optimized TVM TIR / TVMScript code",
```

```
      "status": "string - always 'improved' (you must make at least one optimization change; never return
      ↪  'original')",
      "modification": "string - short description of what was changed"
    }
  - No additional fields or text outside this JSON.

response_format:
  type: json_schema
  json_schema:
    name: tvm_tir_transform
    strict: true
    schema:
      type: object
      properties:
        code:
          type: string
        status:
          type: string
          enum: ["improved"]
        modification:
          type: string
      required: [code, status, modification]
      additionalProperties: false
```

M.4 ITERATION-SUMMARY SYSTEM PROMPT

```
system: |
  You are a summary agent for multi-step TIR optimization. After each step, you write a brief summary of what
  ↪  happened.

  ## Purpose
  This summary will be shown to:
  - the Analysis agent (recommendations)
  - the Code agent (writes TIR code)

  Both must see what was tried and what went wrong so they do not repeat the same mistakes on the next steps.

  ## Input
  You receive:
  - Previous steps summary (concatenated history)
  - Recommendations given for this step
  - The code that was produced
  - Correctness result (if run)
  - Profiling result (latency, profile text, if run)
  - Any error (parse, TIR parse, build, correctness)

  ## Output
  Produce a short, structured summary for this step only. Use this format:

  **Step N:** [What was recommended / attempted]. [Outcome: SUCCESS (latency X ms) or ERROR: <brief
  ↪  description of the error>]. [If error: main cause so agents can avoid it next time]

  ## Rules
  - Write in plain text, no markdown.
  - One step per output; do not summarize previous steps, only the current one.
  - If there was an error, state it clearly and briefly (e.g. "Parse error: invalid TIR", "Build failed:
  ↪  Cannot add loops on root block", "Correctness failed: max_abs_diff=...").
  - If the error suggests a wrong pattern was applied: mention it (e.g. "Wrong pattern: tree reduction was
  ↪  used for elementwise op -- this op has no sum over axes, use direct thread binding + write instead").
  - If success: mention latency and, if useful, what change helped.
  - Be concise (1-4 sentences).
```

N CUDA AGENT PROMPTS AND REPRODUCIBILITY SETTINGS

The CUDA pipelines use a coder together with an outcome-dependent advisor or diagnoser. The advisor is invoked only after a correct candidate and proposes a hardware-specific performance optimization. The diagnoser is invoked after compilation or correctness failure and returns a three-line root-cause analysis. The prompts below are the templates used by the implementation. Jinja fields are retained to show exactly which reference code, candidate, history, measurements, and profiler records are injected.

N.1 ADVISOR PROMPT

```
You are a **CUDA kernel performance advisor**. Another model will turn your advice into `<cuda_source>` /
↪  `<forward_body>`; do **not** emit those tags yourself.

The previous kernel was **correct**. Your only job is to suggest concrete speed-ups for this hardware.
Do **not** re-diagnose correctness issues -- the kernel already passes validation.

**HARD CONSTRAINT -- forbidden ATen shortcuts in `<cuda_source>`:**
The coder is NOT allowed to call any of the following from inside `<cuda_source>` C++ code:
`torch::mm`, `torch::bmm`, `torch::matmul`, `torch::addmm`, `torch::addmv`, `torch::baddbmm`,
`torch::conv1d`, `torch::conv2d`, `torch::conv3d`,
`torch::conv_transpose1d`, `torch::conv_transpose2d`, `torch::conv_transpose3d`,
`torch::einsum`, `torch::linalg::*`, `torch::nn::functional::*`, `at::mm`, `at::matmul`.
Every such call is **rejected at assembly time**. If the reference uses Conv/Matmul/GroupNorm etc., your plan
↪  must describe a hand-written CUDA implementation -- no ATen shortcuts.

{% include '_kernel_feedback_blocks.md.jinja' %}

## Response

Write exactly 2-4 sentences of plain text. No markdown, no bullet points, no headers, no code fences. Propose
↪  one concrete optimization technique (e.g. shared-memory tiling, warp reduction, vectorized loads, loop
↪  unrolling, fused epilogue) and explain why it fits this kernel and hardware. Cite specific kernel names or
↪  values from the profile if present. Start directly with the optimization -- no preamble.
```

N.2 DIAGNOSER PROMPT

```
You are a CUDA debugging expert. Diagnose why the kernel below is broken. Be concise and factual -- no
↪  preamble, no markdown headers, no code blocks.

Reference model (expected behavior):
{{ reference_code }}

Broken kernel:
{{ last_code }}

Error:
{% if not last_result.compiled %}BUILD: {{ (last_result.compilation_error or 'no details')[:800] }}
{% elif not last_result.valid %}CORRECTNESS: {{ (last_result.correctness_info or 'no details')[:800] }}
{% endif %}

Reply with exactly three lines:
ROOT CAUSE: <one sentence -- the fundamental bug>
DETAILS: <two or three concrete facts -- cite variable names or line content from the broken kernel; cover
↪  indexing, dtype, missing ops, nan/inf sources, forward_body args mapping>
FIX: <one sentence -- minimal change needed>

No other text. No bullet points. No markdown.
```

N.3 STRUCTURED FEEDBACK FORMAT

The shared feedback template records the reference, prior hypotheses and outcomes, best-so-far candidate, latest candidate, timing statistics, and the hottest profiled CUDA kernels. Outcomes are routed into four explicit cases, compilation failure with compiler output, incorrect output with validation details, correct output for which timing failed or timed out, and correct output with mean latency, standard deviation, trial count, and profiler summary.

```
{% if hardware_info %}
**Target Hardware:**
{{ hardware_info }}{% if compute_capability %} (Compute Capability: {{ compute_capability }}){% endif %}
{% endif %}

<reference>
{{ reference_code }}
</reference>

{% if history %}
**Previous attempts (oldest first):**
{% for entry in history %}
{% if entry.advisor_text %}
- **Prior advisor plan:** {{ entry.advisor_text[:1200] }}{% if entry.advisor_text|length > 1200 %}...{% endif
↪  %}

{% endif %}
- **Hypothesis:** {{ entry.hypothesis or '(none)' }}
```

```
  **Result:** {% if not entry.result.compiled %}Compilation failed: {{ (entry.result.compilation_error or
  ↪  '')[:500] }}{% elif not entry.result.valid %}Incorrect: {{ (entry.result.correctness_info or '')[:500]
  ↪  }}{% elif entry.result.performance and entry.result.performance.mean %}Correct, mean time: {{
  ↪  "%.3g"|format(entry.result.performance.mean) }}ms{% if entry.result.performance.std %} (± {{
  ↪  "%.3g"|format(entry.result.performance.std) }}ms){% endif %}{% elif entry.result.performance_info
  ↪  %}Correct but perf failed: {{ entry.result.performance_info[:100] }}{% else %}Correct (no perf
  ↪  measurement){% endif %}

  {% if entry.result.valid and entry.result.cuda_profile %}**Hottest kernel:** `{{
  ↪  entry.result.cuda_profile[0].name }}` ({{ "%.2f"|format(entry.result.cuda_profile[0].self_cuda_us /
  ↪  1000.0) }}ms){% endif %}

{% endfor %}
{% endif %}

{% if best_code and best_result %}
**Best attempt so far:**
<best_attempt>
{{ best_code }}
</best_attempt>

**Best Result:**
{% if not best_result.compiled %}
Compilation failed: {{ (best_result.compilation_error or '')[:500] }}
{% elif not best_result.valid %}
Incorrect: {{ (best_result.correctness_info or '')[:500] }}
{% elif best_result.performance and best_result.performance.mean %}
Correct, mean time: {{ "%.3g"|format(best_result.performance.mean) }}ms{% if best_result.performance.std %} (±
↪  {{ "%.3g"|format(best_result.performance.std) }}ms){% endif %}
{% elif best_result.performance_info %}
Correct but perf failed: {{ best_result.performance_info[:200] }}
{% else %}
Correct (no perf measurement)
{% endif %}

{% if best_result.valid and best_result.cuda_profile %}
**Best Kernel Profile ({{ best_result.profiling_mode or 'full' }} mode):**
{% for k in best_result.cuda_profile %}
- `{{ k.name }}`: {{ "%.2f"|format(k.self_cuda_us / 1000.0) }}ms, {{ k.count }}x calls{% if k.flops %}, ~{{
↪  k.flops }} FLOPs{% endif %}
{% endfor %}
Focus optimization efforts on the slowest kernels.
{% endif %}
{% endif %}

{% if last_code and (not best_code or last_code != best_code) %}
**Latest attempt:**
<latest_attempt>
{{ last_code }}
</latest_attempt>

**Latest Result:**
{% if not last_result.compiled %}
Compilation failed:
{{ (last_result.compilation_error or '')[:4000] }}{% if last_result.compilation_error and
↪  last_result.compilation_error|length > 4000 %} ... (truncated){% endif %}
{% elif not last_result.valid %}
Incorrect results:
{{ (last_result.correctness_info or '')[:4000] }}{% if last_result.correctness_info and
↪  last_result.correctness_info|length > 4000 %} ... (truncated){% endif %}
{% elif last_result.performance and last_result.performance.mean %}
Correct, mean time: {{ "%.3g"|format(last_result.performance.mean) }}ms{% if last_result.performance.std %} (±
↪  {{ "%.3g"|format(last_result.performance.std) }}ms){% endif %}{% if last_result.performance.num_trials %},
↪  {{ last_result.performance.num_trials }} trials{% endif %}
{% elif last_result.performance_info %}
Correct but performance measurement failed or timed out: {{ last_result.performance_info }}
{% else %}
Correct (no perf measurement)
{% endif %}

{% if last_result.valid and last_result.cuda_profile %}
**Latest Kernel Profile ({{ last_result.profiling_mode or 'full' }} mode):**
{% for k in last_result.cuda_profile %}
- `{{ k.name }}`: {{ "%.2f"|format(k.self_cuda_us / 1000.0) }}ms, {{ k.count }}x calls{% if k.flops %}, ~{{
↪  k.flops }} FLOPs{% endif %}
{% endfor %}
{% endif %}
{% endif %}
```

For the Sampling Agent, each of the five candidate–environment pairs is first converted into exactly one actionable sentence by the following post-processing prompt. Candidate 6 receives the five code, environment-response, and advice triples together. The Feedback Loop instead sends the corresponding advisor or diagnoser output directly to the next coder call.

```
You review **one** CUDA kernel candidate. You see the reference task, this candidate's generated code, and the
↪  **environment (bench) response** -- compilation outcome, correctness, timings, profiler hotspots when
↪  present.
```

```
Output **exactly one English sentence**: the single most important fix, correction, or performance improvement
↪  for **this** candidate only. No bullet points, no numbering, no quotes around the sentence, no preamble or
↪  closing.

<reference_task>
{{ reference_code }}
</reference_task>

<generated_code>
{{ generated_code }}
</generated_code>

<environment_response>
{{ feedback_text }}
</environment_response>

One sentence only:
```

The CUDA diagnoser uses temperature 0.7 and at most 400 output tokens. The advisor uses temperature 0.7 and at most 512 output tokens. Coder calls use temperature 0.1, top-$p = 0.95$, and at most 16,384 output tokens.

O Benchmark Input-Size Variants

Table 11 lists the tensor shapes used in the Base and Small regimes for five example tasks. Further details are available in the accompanying repository. We use the following notation. M and N are the output matrix dimensions (rows and columns), K is the shared inner dimension of a matrix product, *batch* is the batch size, *dim* is a feature or reduction dimension, C_{in} and C_{out} are the input and output channel counts of a convolution, H and W are spatial height and width, k is the convolution kernel size, and *in*, *hidden*, *out* are the input, hidden, and output feature dimensions of a linear layer.

Level	Task	Base	Small
L1	Standard matmul	$M = 2048$ $K = 8192$ $N = 4096$	$M = 128$ $K = 512$ $N = 256$
L1	Softmax	batch $= 4096$ dim $= 393216$	batch $= 4$ dim $= 384$
L2	Conv2D + ReLU + bias add	batch $= 128$ $C_{\text{in}} = 64$ $C_{\text{out}} = 128$ $H = W = 128$ $k = 3$	batch $= 32$ $C_{\text{in}} = 16$ $C_{\text{out}} = 32$ $H = W = 32$ $k = 1$
L2	Gemm + Sigmoid + LSE	batch $= 16384$ in $= 2048$ hidden $= 4096$ out $= 1024$	batch $= 512$ in $= 64$ hidden $= 128$ out $= 32$
L2	Gemm + Swish + Divide + clamp stack	batch $= 1024$ in $= 8192$ out $= 8192$	batch $= 64$ in $= 512$ out $= 512$

Table 11: Tensor shapes for five example tasks in the Base and Small regimes. Values are taken directly from the matched `level1`/`level1-small` and `level2`/`level2-small` task definitions. Task labels are shortened for readability. Variable notation is defined in the text above.

P Agentic Pipeline Diagrams

Figure 12 illustrates the pipeline of each agentic architecture from two complementary perspectives, a flowchart view and a graph-structure view. Table 12 summarizes the key properties of both architectures.

Q Full Feedback-Iteration Breakdown

Figure 13 expands Figure 6 from the first and last feedback iterations to the complete five-step trajectory.

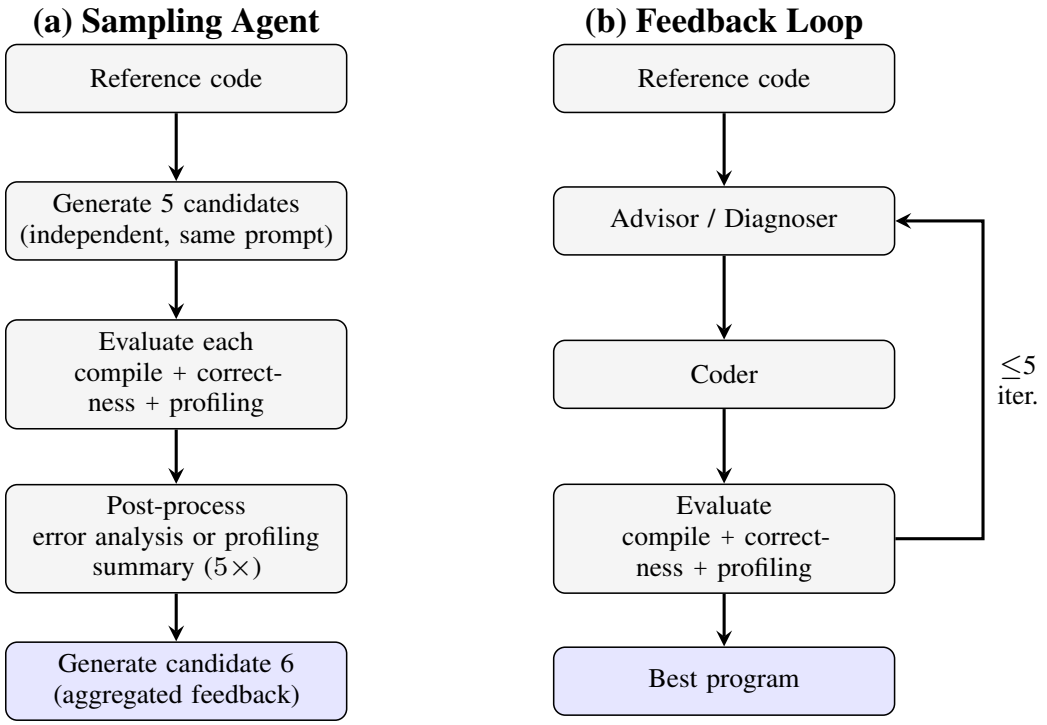

Figure 12: Pipeline diagrams for the two agentic architectures. **(a)** *Sampling Agent*. Five candidates are generated independently from the reference and evaluated. Post-processing converts each outcome into structured feedback, which is aggregated to generate a sixth candidate. **(b)** *Feedback Loop*. An advisor (on success) or diagnoser (on failure) iteratively steers a coder stage for up to five iterations, with the original reference and current best always in context.

Table 12: Summary of the two agentic architectures used in Section 4.1. #LLM is the maximum number of model calls per task. The Sampling Agent uses 5 coder + 5 advisor/diagnoser + 1 coder and the Feedback Loop uses 5 coder + 5 advisor/diagnoser.

Agent	Representation	#LLM	Search strategy	Feedback per step
Sampling Agent	CUDA, TVM IR	11	Parallel sampling, 6th candidate conditioned on 5 aggregated feedbacks	Error analysis or profiling summary (5×)
Feedback Loop	CUDA, TVM IR	10	Sequential refinement, advisor or diagnoser selected by outcome	Advice, profiling, and best-so-far in context

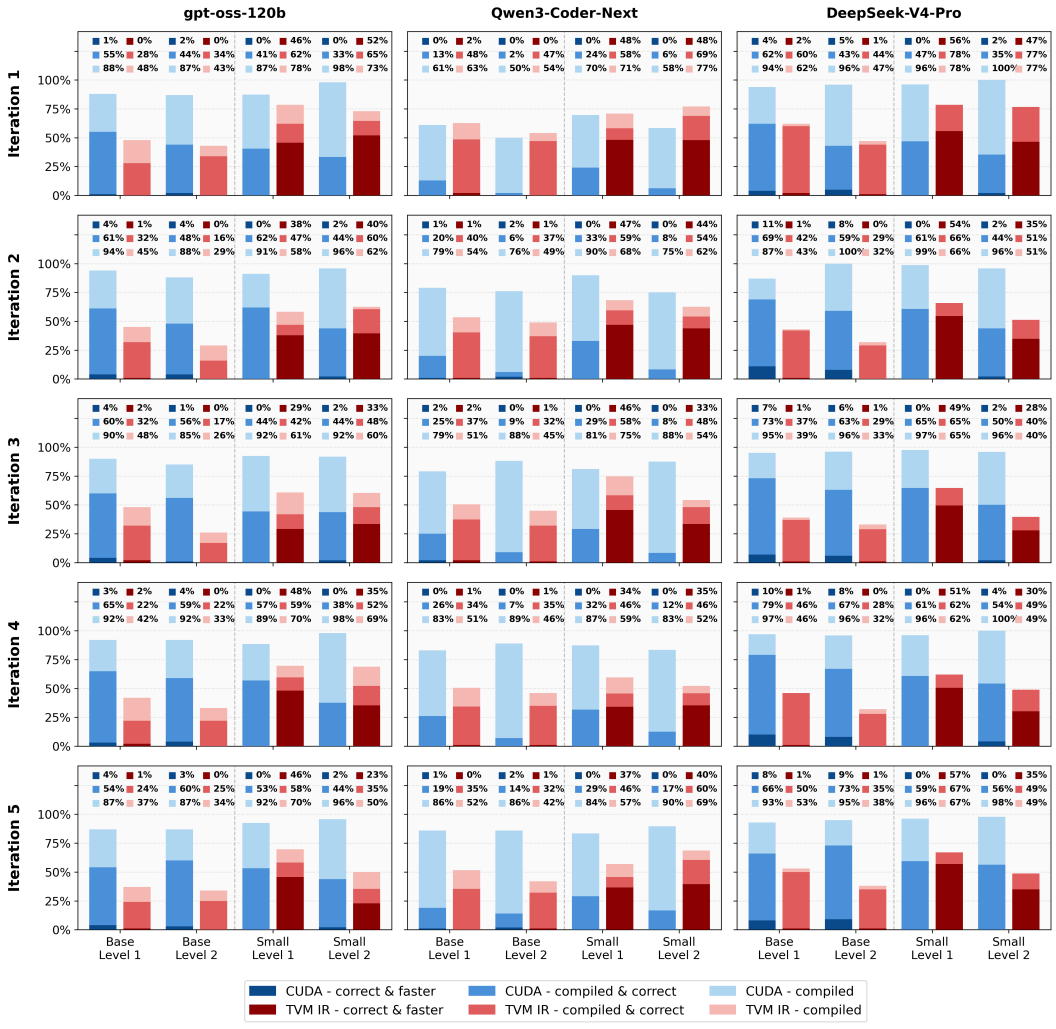

Figure 13: Expanded Feedback Loop validity and speedup under CUDA and TVM IR generation across all five feedback iterations. Bars use the same nested compile, correctness, and speedup encoding as the main-text figure. Panels cover `gpt-oss-120b`, `Qwen3-Coder-Next`, and `DeepSeek-V4-Pro`.

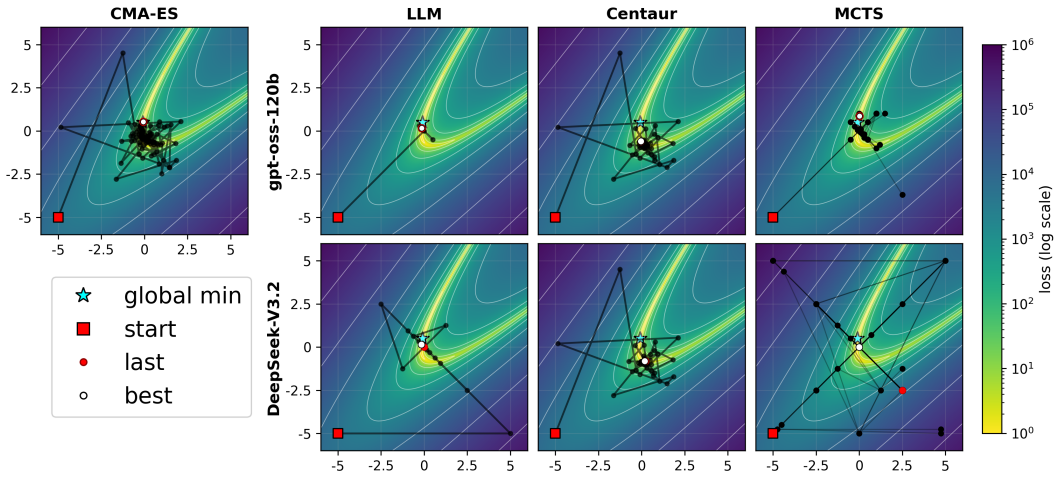

Figure 14: BBOB task `bbob_f09_rosenbrock_rotated_i1`.

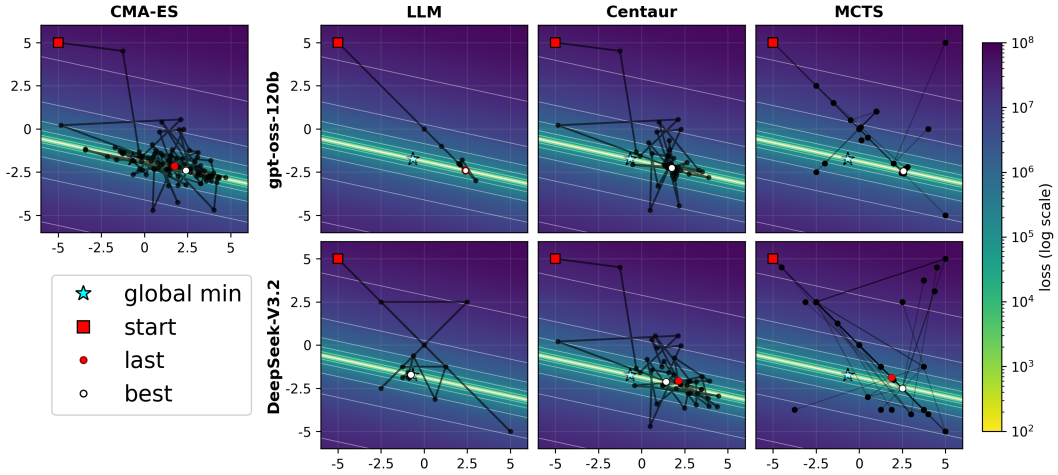

Figure 15: BBOB task `bbob_f10_ellipsoid_rotated_i1`.

R Additional BBOB Optimization Traces

Figures 14-18 show additional BBOB optimization traces for `gpt-oss-120b` and `DeepSeek-V3.2`. Filenames match BBOB task identifiers (`bbob_fXX_..._iY`).

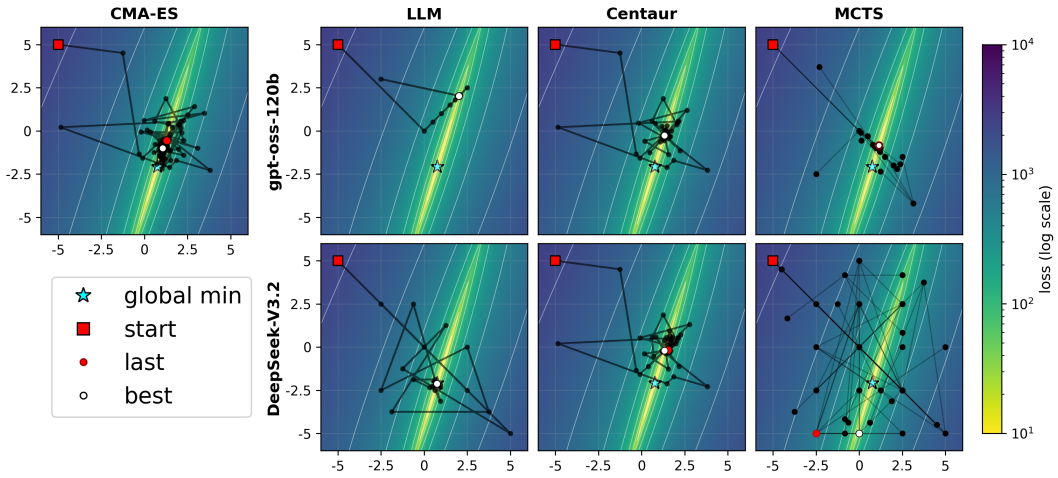

Figure 16: BBOB task `bbob_f13_sharp_ridge_i1`.

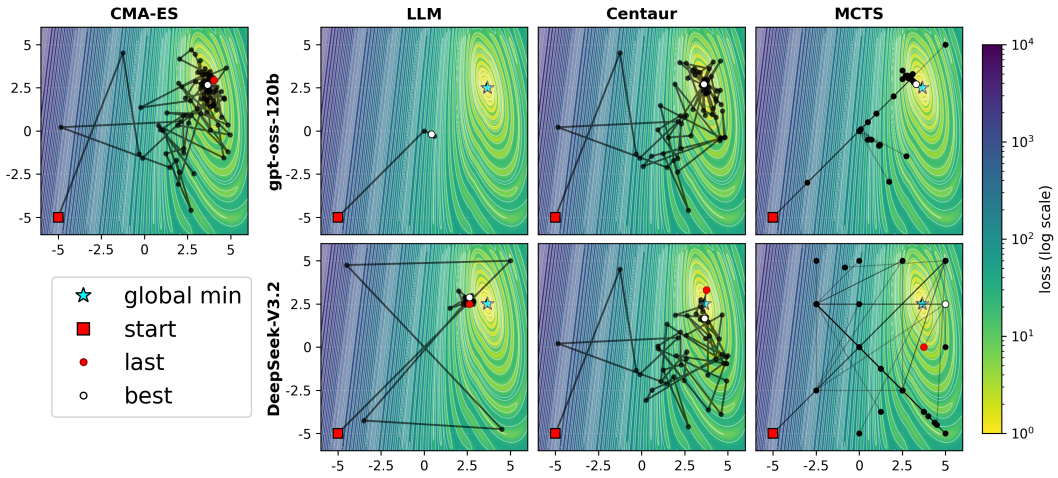

Figure 17: BBOB task `bbob_f17_schaffers_f7_i1`.

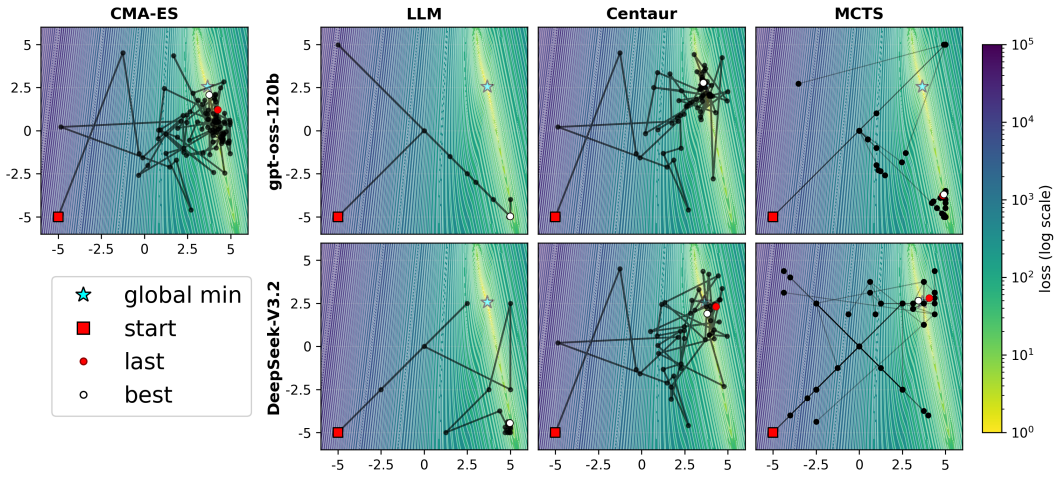

Figure 18: BBOB task `bbob_f18_schaffers_f7_ill_i1`.