How to Train Your Little Dragon: Teaching a Small Language Model to Construct and Refine GPU Optimization Scripts

Anonymous authors
Anonymous working draft

Abstract

Compiler transformations interact, making GPU kernel optimization a search for combinations that preserve the computation while improving performance. We investigate how to train a small language model to generate the optimization: a correct, performance-improving script that coordinates interacting transformations in one complete proposal. We introduce How to Train Your Little Dragon, a training method that teaches a small language model both to construct complete optimization scripts and to improve existing ones. Rather than learn only to rank search candidates or generate replacement kernel code, our Little Dragon learns to generate correct scripts that transform the under-optimized kernel and improve its performance. We devise a compact, lossless representation whose deterministic decoding reconstructs the exact compiler trace, preserving its transformation parameters and dependencies. We use a compiler framework TVM MetaSchedule to build training data on 54,116 distinct workloads, and construct 108,594 examples in two training sets: 60,236 teach the model to generate an optimized script from an unoptimized kernel, while 48,358 teach it to produce a better script given the kernel and a somewhat optimized script. Supervised fine-tuning teaches the model to construct and improve optimization scripts from these search-derived examples. We then add reinforcement learning with a shaped reward that combines correctness with measured performance, awarding a latency-based bonus only to numerically correct kernels. This enables the model learn not only from existing optimizations, but from the correctness and performance of its generated scripts. We apply this training method to a model with 8 billion parameters. After supervised fine-tuning and reinforcement learning, the model achieves per-attempt correctness of 91.27% for construction and 92.01% for refinement, while reinforcement learning produces faster kernels than supervised training on 88 of 102 construction tasks and 145 of 170 refinement prompts. With only 8 billion parameters, our Little Dragon achieves higher per-attempt correctness and better kernel performance than six frontier LLMs—GPT-6 Astra, Opus 5, Kimi-K3, and GLM-5.3 among them, each with hundreds of billions to trillions of parameters—for which task-specific training is unavailable or expensive. These results suggest that How to Train Your Little Dragon enables a small language model to learn from compiler search and direct correctness and performance feedback, internalizing an optimization policy that constructs and improves complete scripts in one shot.

1 Introduction

The cost of serving a model depends not only on the model itself, but on how efficiently its kernels use the GPU. Compiler optimization can reduce this cost by turning the same computation into a faster implementation, yet finding that implementation can itself require substantial compilation and measurement. The difficulty is that transformations interact: tiling, memory placement, and thread mapping must work together to preserve correctness and improve performance. Search-based compilers discover effective combinations through trial and error, repeatedly generating and evaluating candidates (Zheng et al., 2020; Shao et al., 2022). We ask how to train a small language model to construct and improve complete optimization scripts in one shot, learning to coordinate interacting compiler transformations into correct, faster kernels.

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.

Existing approaches use learning at different points in optimization. Learned cost models rank candidates for evolutionary search, while frozen language models guide exploration without updating their parameters (Chen et al., 2018b; Tang et al., 2025; 2026). Other approaches train models to generate scheduling decisions step by step, conditioning each choice on earlier decisions while its benefit may depend on transformations yet to be selected (Zhai et al., 2024). We set out to explore a research question: can a small language model learn an optimization policy across different tasks that enables it to construct and improve complete optimization scripts in one shot, and how can we train it to do so?

Our hypothesis is that learning to improve an optimization requires examples not only of effective solutions, but of how less effective solutions can become better. We therefore introduce HOW TO TRAIN YOUR LITTLE DRAGON, a training method built around two complementary sets of examples. In the first, we give the model an unoptimized kernel and teach it to generate an optimized script. In the second, we give it the unoptimized kernel together with a somewhat optimized script and teach it to generate a more optimized script. Both tasks produce a complete script, allowing refinement to revise earlier and later transformation choices together rather than merely extend the existing sequence. Through examples from different optimization tasks, we seek to teach the model which combinations work well together and how to improve them, internalizing an optimization policy that it can apply to other kernels.

The supervised examples teach correct, performance-improving optimizations. We then use reinforcement learning to strengthen the policy through feedback on the correctness and measured performance of its own complete proposals. A staged reward recognizes progress toward executable, correct kernels and awards latency bonuses only to numerically correct candidates, with refinement bonuses additionally requiring improvement over the supplied script.

Both stages require the model to express its complete optimization precisely enough for the compiler to apply it. A script must specify not only which transformations to use, but how they refer to the loops and memory structures created by earlier transformations. Raw compiler traces preserve this information through detailed parameters and temporary identifiers, while prose descriptions can explain an optimization without specifying everything needed to reproduce it. We therefore devise a compact, lossless representation that groups related transformations into typed edits while retaining their order, parameters, and dependencies. Deterministic decoding reconstructs the exact canonical compiler trace, and round-trip checks verify that representing a source script does not change it. The model is taught explicit compiler optimization decisions connected to their supporting program evidence. Optimization graphs link compiler-derived facts, optimization analysis, executable schedule edits, and expected effects, exposing both how an optimization is applied and why it is proposed. Search-derived traces fix the executable targets, while static checks ground the accompanying explanations in compiler artifacts. This provides a structured teaching interface for learning compiler optimization through SFT and execution feedback.

To supply examples in this representation, we use TVM MetaSchedule, a compiler framework that explores candidate optimizations through stochastic evolutionary search guided by performance cost models (Shao et al., 2022). From this search, we select the scripts that produce the best-performing validated kernels as training targets. We pair each target with the unoptimized kernel to teach construction, and additionally supply a somewhat optimized script to teach refinement. For refinement, we require the target's measured improvement to exceed timing variability, so the example reflects a genuine performance gain. Across 54,116 distinct workloads, we construct 108,594 training examples: 60,236 for construction and 48,358 for refinement. Thus, we turn the search process into a source of both correct optimization scripts and concrete examples of how to improve them.

We evaluate whether this training enables a small model to generate correct optimization scripts and improve the performance of the resulting kernels. We evaluate a model with 8 billion parameters on NVIDIA H100. After supervised fine-tuning and reinforcement learning, the model achieves per-attempt correctness (Correct@1) of 91.27% for construction and 92.01% for refinement. Comparing each model's fixed first output, reinforcement learning produces faster kernels than supervised training on 88 of 102 construction tasks and 145 of 170 refinement prompts for which both outputs are correct and timed. With only 8 billion parameters, HOW TO TRAIN YOUR LITTLE DRAGON achieves higher per-attempt correctness and better kernel performance than six frontier LLMs—GPT-6 Astra, Opus 5, Kimi-K3, and GLM-5.3 among them—for which task-specific training would be unavailable or expensive. These results provide an affirmative answer to our research question and a concrete way

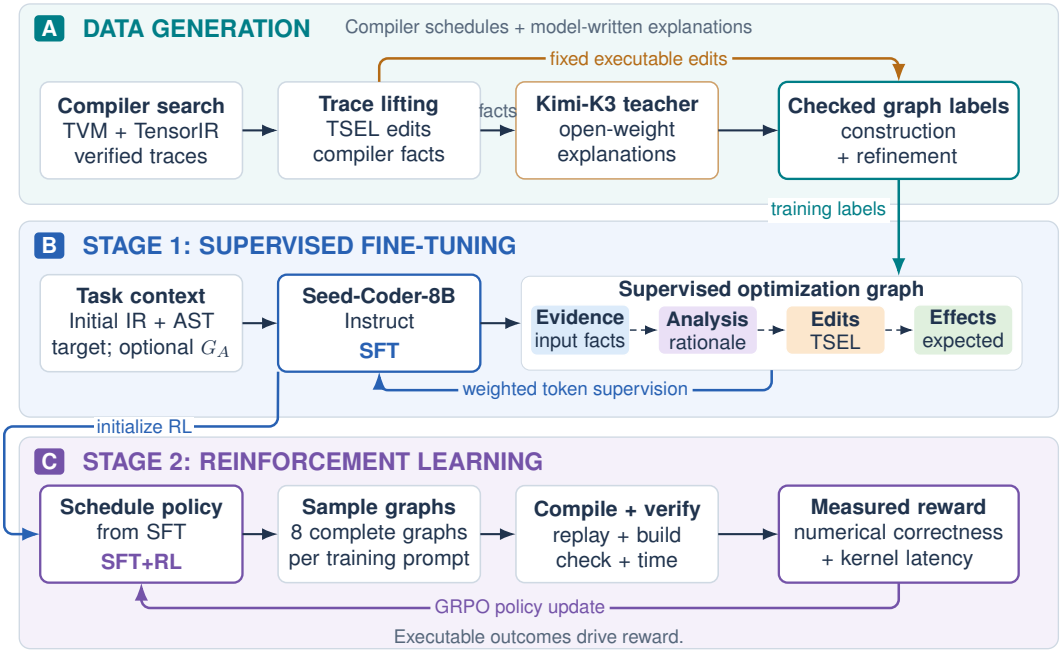

Figure 1: **How to Train Your Little Dragon: data generation and two-stage policy training.** **(A)** Compiler search supplies verified schedules. Deterministic lifting preserves their executable edits in our Typed Schedule Edit Language (TSEL); Kimi-K3 writes explanations from compiler-derived facts. Static and round-trip checks admit construction and refinement labels. **(B)** Weighted token supervision trains Seed-Coder-8B-Instruct to produce complete optimization graphs; G_A denotes the optional prior graph for refinement. **(C)** Starting from SFT, GRPO updates the policy using numerical correctness and measured latency of sampled schedules.

to achieve it: training on construction and refinement examples, followed by rewards for correctness and performance, enables a small model to internalize an optimization policy for generating complete scripts in one shot.

2 Learning Executable Compiler Transformations

How to Train Your Little Dragon separates three responsibilities: the compiler defines and executes transformations, the policy chooses a complete schedule, and a verifier evaluates its correctness and performance. Figure 1 separates training-data generation, supervised fine-tuning, and execution-grounded reinforcement learning.

2.1 Optimization Task and Graph Interface

Apache TVM, an open-source deep learning compiler, constructs the unscheduled module M_0 in TensorIR, its loop-level intermediate representation, through *lowering* (operators to loops and memory accesses) and *task extraction* (isolating separately tunable operator groups). Rule-based analysis of its abstract syntax tree (AST) deterministically extracts structural features $F(M_0)$: loop extents, buffer shapes, reduction axes, and block structure. The policy receives $x = (d, M_0, F(M_0), h)$, with textual task description d and compiler environment h (target hardware and compiler configuration). Appendix A.1 gives a concrete example.

A *schedule* specifies compiler transformations such as loop tiling and GPU thread mapping. Its *trace* τ records primitive calls, arguments, and sampling decisions:

$$M_\tau = \mathrm{Replay}_h(M_0, \tau), \qquad K_\tau = \mathrm{Compile}_h(M_\tau). \tag{1}$$

Here Replay deterministically applies the trace, rejecting illegal transformations; $\mathrm{Compile}$ generates executable kernel K_τ from transformed module M_τ. The subscript h marks environment dependence. We seek a trace whose kernel has low measured latency $L_h(K_\tau)$ on the target and passes a numerical

comparison with the original computation. Refinement augments x with an existing candidate graph G_A and a summary of its schedule. The intended output in both modes specifies a complete trace rooted at M_0. A refined trace replaces τ_A; it is not appended to the already transformed program.

The policy output $y = (y_1, \ldots, y_T)$ is the token sequence serializing a complete optimization graph $G = (V, E, P)$. Here T is the output length, y_t is its t-th token, and $y_{<t}$ is the preceding output prefix. The graph's nodes V have four roles: evidence quotes facts from the input; analysis relates these facts to optimization opportunities; edits specify compiler actions in our *Typed Schedule Edit Language* (TSEL); and expected effects describe anticipated consequences. Typed edges E connect evidence to analysis (`supports`), analysis to edits (`motivates`), and edits to effects (`predicts`). The payload P orders edit groups and the instructions inside them. Replay follows this executable order, not a traversal of the explanatory edges. This separation allows exact execution and semantic explanation to be checked independently. Figure 2 illustrates this separation with an actual projection training label: an edit's serialized payload determines its tiling factors, while E provides explanatory links and P fixes replay order.

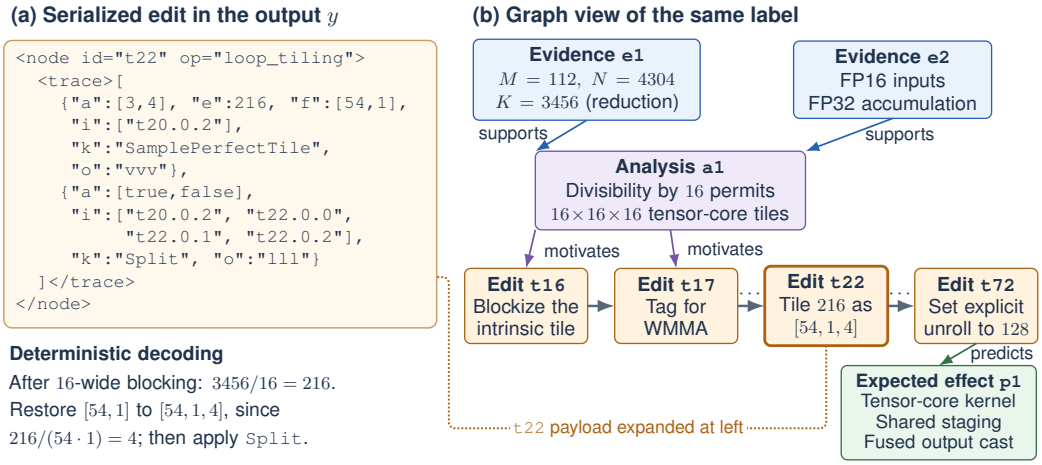

Figure 2: **A concrete optimization graph.** The payload fragment and graph projection come from the same FP16 projection training label. Node text is shortened; ellipses omit other edits. Colored edges E explain choices; gray arrows show replay order P, and the dotted callout expands `t22`. WMMA denotes tensor-core operations. The expected effect `p1` concerns the whole schedule.

2.2 Compact and Lossless Schedule Edits

TSEL represents each semantic edit as an ordered fragment of the TVM trace. For example, a tiling edit combines a sampling instruction with the split that consumes its factors; a memory-hierarchy edit introduces a cache stage. Grouping makes these actions explicit without replacing their executable parameters with prose.

TSEL retains instruction kinds, typed inputs and outputs, attributes, sampling decisions, and the postprocessing boundary. Instead of global temporary-variable counters, a reference identifies its producer by *edit group, local instruction, and output slot*. Index maps use a versioned, typed expression DAG, with shared maps stored once in a pool. The decoder reconstructs global handle numbering and canonical compiler objects. For every admitted source trace, the representation satisfies

$$\mathrm{Canon}_h(\mathrm{Lower}_h(\mathrm{Lift}_h(\tau))) = \mathrm{Canon}_h(\tau). \tag{2}$$

This round-trip condition preserves executable information while eliminating redundant serialization. It is checked under the recorded compiler and codec versions; it is not a claim of equivalence across arbitrary TVM releases.

The codec can also derive a redundant final tile factor. For loop extent e and a factorization of arity n, it stores e and $f_1, \ldots, f_{n-1}$ and reconstructs

$$f_n = \frac{e}{\prod_{j=1}^{n-1} f_j}, \tag{3}$$

rejecting nonintegral or otherwise illegal results. Reference resolution and this arithmetic check do not replace schedule replay: an edit must still be applicable to the program state created by its predecessors. Figure 2 shows a serialized instruction fragment from a training label. Appendix A provides the corresponding policy input and further details of the representation.

2.3 Supervision from verified search

Compiler search provides records (M_0, τ, M_τ, L) together with AST features and verification outcomes. We retain candidates that replay, compile, pass numerical checks, and have stable measured latency. Construction selects a fast candidate for each input. Refinement pairs a slower verified candidate with a faster candidate of the same computation, requiring the measured gap to exceed the prescribed reproducibility margin. Both endpoints retain their own complete traces; pairing does not assume that one trace is a valid incremental suffix of the other.

Label generation is split between deterministic tooling and a teacher model. The tooling lifts the chosen trace, derives facts about its effects on program structure, and fixes the exact edit payload. The teacher receives these facts and writes evidence, analyses, edit summaries, and expected effects. A static verifier checks cited sources, arithmetic derivations, applicability statements, and their consistency with the executable actions. Thus, an imprecise teacher explanation cannot silently change the transformation being taught.

The student receives only information available for its task: the initial program and features, and, for refinement, the supplied prior schedule and its summary. Target-side IR and measured target latency are kept out of the student prompt. The implemented refinement interface also includes an improvement-band request. Explanations provide supervised descriptions of schedule choices, not certificates that a particular edit caused a measured speedup.

2.4 Supervised and execution-grounded policy learning

SFT minimizes weighted autoregressive negative log-likelihood over graph completions:

$$\mathcal{L}_{\mathrm{SFT}} = -\frac{\sum_{(x,y)} \sum_t w_t \log \pi_\theta(y_t \mid x, y_{<t})}{\sum_{(x,y)} \sum_t w_t}. \tag{4}$$

Consistent with prior SFT work on nonuniform token supervision (Kim et al., 2025; Li et al., 2026), we use fixed weights $w_t = 1$ for evidence, analysis, and expected-effect node spans; $w_t = 4$ for sampled-decision payload spans, such as tiling-factor values, which directly govern schedule validity and kernel performance; and $w_t = 2$ for all remaining output tokens, including trace structure and the end-of-sequence token. These spans are annotated deterministically before tokenization. A token crossing span boundaries receives the minimum weight of the characters it covers. Prompt tokens are masked out; in refinement dialogues, the previous assistant graph is context and only the final answer contributes to the loss. Thus ordinary executable content and sampled decisions receive twice and four times the weight of explanation-node content, respectively.

RL samples candidate graphs and executes their recovered traces. The final verification outcome determines the staged reward:

$$R_{\mathrm{stage}} = \begin{cases} 0, & \text{worker crash or unparseable payload,} \\ 0.05, & \text{parsed, but validation or lowering fails,} \\ 0.05 + 0.05 \min(\max(f_{\mathrm{div}}, p), 0.99), & \text{lowered, but replay fails,} \\ 0.15, & \text{replayed, but compilation fails,} \\ 0.35, & \text{compiled, but numerically incorrect,} \\ 1 + r_{\mathrm{lat}}, & \text{numerically correct.} \end{cases} \tag{5}$$

Here f_{div} is the emitted-tile divisibility fraction, p is the replayed-instruction fraction, and r_{lat} is the latency bonus for a numerically correct candidate. We define numerical correctness as successful trace replay and compilation followed by output agreement with the LLVM-compiled original TensorIR using one randomly generated test case per workload. Intermediate rewards distinguish progress through parsing, replay, and compilation, providing learning signals for partially successful proposals, while latency bonuses require numerical correctness, consistent with previous work (Dai et al., 2026).

To compare latency gains across workloads, let L_0 be the default-scheduled latency and $a < L_0$ a fixed reference anchor. We measure kernel latency as the median of five synchronized timing repeats after warmup, and calibrate task-specific noise bands by benchmarking identical schedules in three separate evaluator runs, each launched in a fresh process. Define the normalized utility

$$u(L) = \frac{\log(L_0/L)}{\log(L_0/a)}, \qquad Q_m(u) = q_m \lfloor u/q_m \rfloor , \tag{6}$$

where q_m is a noise-dependent quantum for task mode m. Quantization prevents small timing fluctuations from always producing a distinct reward. The latency bonus in eq. (5) is

$$r_{\text{lat}} = \alpha_m b_m(L) \operatorname{clip}(Q_m(u(L)), 0, u_{\max}) , \tag{7}$$

Here α_m is the mode-specific latency-reward coefficient: $\alpha_{\text{main}} = 0.30$ for construction and $\alpha_{\text{refine}} = 0.50$ for refinement. The cap is $u_{\max} = 1.2$, and b_m requires a sufficiently large improvement over the default schedule for construction and over the supplied parent for refinement. The latency coefficient is bounded so that ordinary correct candidates outrank incorrect ones.

We optimize this reward with group-relative policy optimization (Shao et al., 2024). Advantages compare sampled candidates for the same input, and zero-variance groups are excluded from updates. The executable payload determines the reward independently of its explanatory text. Partial credit reflects progress through the execution pipeline, while the latency bonus requires numerical correctness and a measured improvement satisfying the prescribed gates. A candidate therefore cannot earn performance reward merely by producing a persuasive explanation or claiming a speedup: its recovered trace must produce a kernel whose measured behavior earns that reward. This separation provides an explicit safeguard against reward hacking through unsupported optimization claims. Details on reinforcement learning optimization policy appear in Appendix B.

3 RESULTS

Following the experimental setup (Section 3.1), we examine four questions. Section 3.2 asks whether search-derived supervision can teach an 8B model to construct and refine optimizations, showing high correctness and substantial kernel speedups on held-out tasks. Section 3.3 tests whether reinforcement learning feedback improves this policy, finding further kernel speedups while maintaining high correctness. Section 3.4 examines whether the trained model can outperform frontier baselines, finding that it leads all six. Section 3.5 compares its optimization cost with TVM search, and the model produces faster kernels with substantially less optimization time.

3.1 EXPERIMENTAL SETUP

Data and training. Our workloads span matrix multiplication, attention, normalization and reductions, mixture-of-experts operations, convolution, and quantization. We fine-tune Seed-Coder-8B-Instruct (ByteDance Seed et al., 2025) for two epochs on 59,040 construction and 47,232 refinement examples, with explanation labels generated by Kimi-K3 (Kimi Team, 2026), then apply execution-grounded RL. Splits hold out entire workload keys and check task names and initial-IR hashes. SFT is selected using validation; the reported RL checkpoint is selected using a post-hoc validation-latency criterion.

Evaluation and baselines. We evaluate on NVIDIA H100 using 116 held-out construction tasks and 194 refinement prompts spanning 39 tasks. We separate examples by workload identity and verify zero overlap between the sealed evaluation tasks and the SFT training, validation, and RL training pools using both workload identities and recorded structural hashes of the initial TensorIR modules. A task is a statically shaped TensorIR computation; a prompt is one model input. Construction supplies one prompt per task, whereas refinement supplies multiple prompts for some tasks, with different parent schedules or refinement requests. Each model generates eight outputs per prompt, totaling 928 construction outputs and 1,552 refinement outputs. Recovered programs are compiled, checked numerically, and timed under the recorded protocol. Compiler references are TVM's default GPU schedule (L_0) and its archived MetaSchedule search results. We compare SFT and SFT+RL with GPT-6 Astra (OpenAI, 2026), Opus 5 (Anthropic, 2026), Seed-Coder-8B-Instruct, Kimi-K3, GLM-5.3 (Z.ai, 2026), and Qwen3.8-27B (Qwen Team, 2026). For construction, external baselines

Model	Construction (%)			Refinement (%)		
	Correct@1	fast_2	fast_{10}	Correct@1	Correct and improves graph A	Correct and $\geq 2\times$ over graph A
GPT-6 Astra	9.38	7.65	6.03	45.55	36.47	17.33
Opus 5	64.01	58.30	36.96	45.81	31.70	14.24
Kimi-K3	34.91	25.43	14.55	37.82	19.59	4.70
GLM-5.3 (Z.ai)	15.73	11.75	7.00	18.56	8.25	1.80
Qwen3.8-27B	0.11	0.00	0.00	19.59	4.57	1.48
Seed-Coder-8B-Instruct	0.00	0.00	0.00	8.12	4.51	1.22
SFT (ours)	89.33	84.16	64.98	91.95	68.56	37.24
SFT+RL (ours)	91.27	89.33	77.69	92.01	87.05	59.28

Table 1: **Per-output correctness and optimization success.** Results cover 116 construction tasks and 194 refinement prompts spanning 39 tasks.

receive PyTorch reference code and generate complete, self-contained TVMScript TensorIR kernels. This interface avoids requiring them to emit our custom optimization-graph representation. For refinement, they generate complete optimization graphs from workload–parent inputs.

Metrics. Correct@1 is the $k{=}1$ case of pass@k (Chen et al., 2021) with numerical correctness as success. We define construction fast_p as the percentage of all generated outputs that are numerically correct and achieve speedup $L_0/L > p$, where L is the candidate latency and L_0 is the task's default-scheduled latency; Table 1 reports fast_2 and fast_{10}. Refinement performance columns instead require correctness and improvement over the supplied parent, graph A: either beyond the task's recorded noise band or by at least $2\times$, using the parent's latency L_A as the reference. Untimed outputs receive no performance credit. Paired SFT–RL speedup-win counts use a separate population: prompts for which both models' fixed first outputs are correct and timed.

3.2 SFT TRANSFERS COMPILER-SEARCH EXPERIENCE TO AN 8B MODEL

Search-derived supervision teaches the small model to generate complete, effective optimizations on held-out tasks. SFT achieves Correct@1 of 89.33% for construction and 91.95% for refinement (Table 1). Each accepted proposal specifies a complete transformation script that replays, compiles, and passes the numerical checks. The performance columns show that this capability includes substantial optimization: SFT achieves construction fast_{10} of 64.98%, while 68.56% of refinement outputs correctly improve the supplied graph A beyond the recorded noise band. These results support both training objectives from the introduction: learning to construct an optimization from an unoptimized kernel and learning to improve an existing optimization. SFT thus supplies a policy that can already produce useful complete scripts, providing the starting point for learning from their outcomes.

3.3 RL IMPROVES SCHEDULE QUALITY THROUGH EXECUTION FEEDBACK

Execution feedback directs an already capable policy toward faster kernels. At the selected RL checkpoint, construction fast_{10} rises from 64.98% to 77.69%. For refinement, the fraction that are correct and at least twice as fast as graph A rises from 37.24% to 59.28% (Table 1). Correctness remains high, increasing from 89.33% to 91.27% for construction and from 91.95% to 92.01% for refinement. The larger changes in performance-success rates show where RL contributes most: it strengthens the model's ability to propose a useful optimization once SFT has established executable generation.

The improvement is visible in the fixed first proposal. Among prompts with correct, timed first outputs from both models, RL produces faster kernels on 88 of 102 construction tasks and 145 of 170 refinement prompts. Together with the threshold-success rates, this supports the introduction's hypothesis that feedback on the model's own complete proposals can improve the optimization policy learned from search-derived demonstrations.

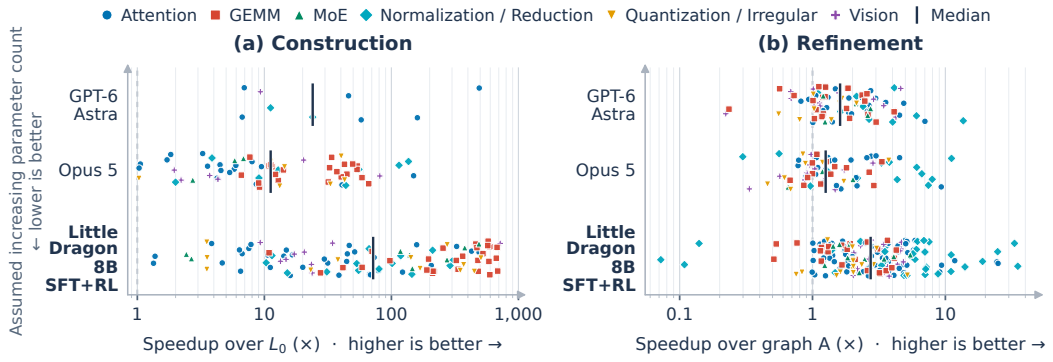

Figure 3: **First output kernel speedups.** Our LITTLE DRAGON achieves better speedups for both the construction and the refinement tasks across multiple kernel famiilies.

3.4 HOW TO TRAIN YOUR LITTLE DRAGON OUTPERFORMS FRONTIER BASELINES

The trained 8B model leads all six reported metrics across construction and refinement. Table 1 compares it with six external baselines, including GPT-6 Astra and Opus 5. The comparison also includes much larger open models: Kimi-K3 has 2.8T total parameters, with 104B activated per token (Kimi Team, 2026), and GLM-5.3 has approximately 753B total parameters (Z.ai, 2026).

Training a small model yields a large correctness advantage. As shown in Table 1, for construction, SFT+RL reaches 91.27% Correct@1, compared with 64.01% for the strongest external baseline, Opus 5—a 27.26 percentage-point advantage. For refinement, the corresponding rates are 92.01% and 45.81%, a 46.20 percentage-point advantage. SFT alone already exceeds every external baseline in all six columns; RL extends that advantage. This progression connects the comparison to the training method: search-derived supervision establishes strong construction and refinement performance, and execution feedback further improves the trained policy.

LITTLE DRAGON improves kernel speedup substantially with a much smaller model size. For construction, SFT+RL achieves fast_{10} of 77.69%, compared with 36.96% for Opus 5, the strongest external model on this metric. For refinement, 59.28% of SFT+RL outputs are correct and at least twice as fast as their supplied graph A, compared with 17.33% for GPT-6 Astra, the strongest external model on that metric. These comparisons retain failures in the denominator, measuring how often one attempt yields a correct kernel with a substantial performance gain. They support the paper's central claim: training on compiler-search experience and execution outcomes enables an 8B model to outperform frontier baselines, including models with hundreds of billions to trillions of parameters, on the reported optimization tasks. Figure 3 complements these success rates with speedup distributions for correct, timed first outputs on the plotted subsets. The trained policy's median speedups are $71.8\times$ over L_0 for construction and $2.74\times$ over graph A for refinement, compared with $11.2\times$/$1.26\times$ for Opus 5 and $24.0\times$/$1.61\times$ for GPT-6 Astra.

3.5 COMPARISON WITH TVM SEARCH

LITTLE DRAGON achieves better kernels with substantially less per-task optimization time. Figure 4 compares optimization time and kernel quality across six workload families between first SFT+RL output and TVM MetaSchedule search run. Under the plotted timing convention, SFT+RL produces these faster kernels with approximately $39\times$ less optimization time, measured by the median paired TVM-to-model time ratio. The family medians range from $24\times$ to $79\times$: $79\times$ for Attention, $42\times$ for GEMM, $36\times$ for MoE, $33\times$ for Normalization/Reduction, $24\times$ for Quantization/Irregular, and $39\times$ for Vision. Kernel speedups use each task's own default schedule, L_0. In the Attention panel, dense MLA decode reaches $44.45\times$ speedup over L_0 with SFT+RL versus $34.25\times$ with TVM, at plotted optimization times of 1.64 versus 170.94 minutes—approximately $104\times$ less time. In the GEMM panel, FP8 matrix multiplication reaches $585.33\times$ versus $373.98\times$ speedup, using 1.24 versus 50.22 minutes, or approximately $41\times$ less time. In the Normalization/Reduction panel, the fused softmax task in a matmul–GELU–softmax workload reaches $70.49\times$ versus $22.18\times$ speedup, using 1.03 versus 28.66 minutes, or approximately $28\times$ less time. These examples show how

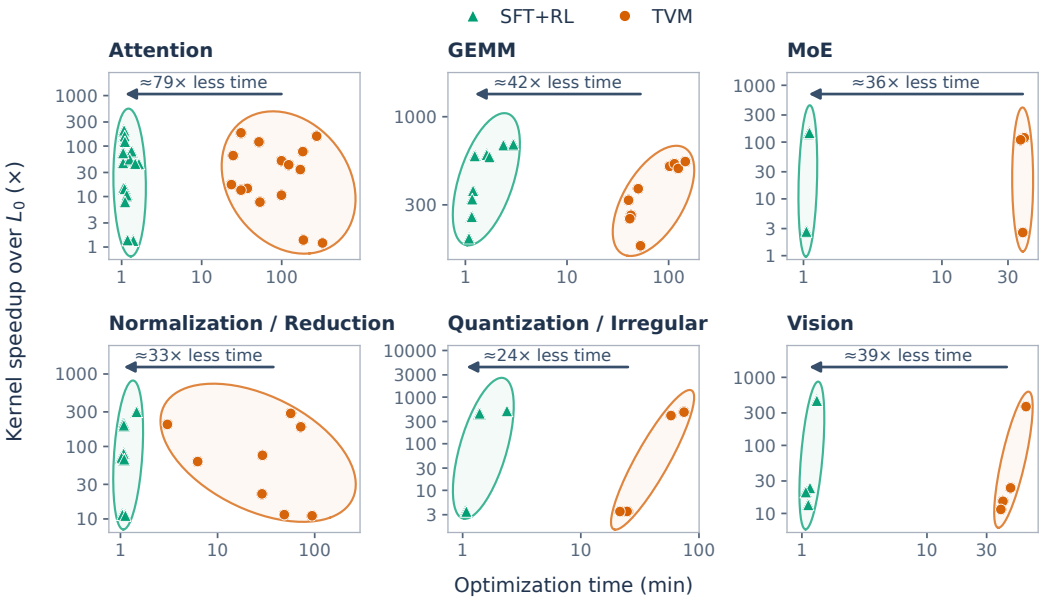

Figure 4: **Kernel speedup and optimization cost comparisons** by family between first SFT+RL output and TVM MetaSchedule search run.

compiler-search experience can be reused through training: the 8B policy produces a complete schedule in a single proposal, reducing the per-task cost of obtaining an optimized kernel.

4 Related Work

Compiler and LLM-guided search. TVM separates computation from scheduling; Ansor and MetaSchedule search transformations (Chen et al., 2018a; Zheng et al., 2020; Shao et al., 2022), while AutoTVM learns cost models to guide measurement (Chen et al., 2018b). Reasoning Compiler and LiteCoOp use LLM-guided search trees (Tang et al., 2025; 2026); AccelOpt, AlphaEvolve, and KernelEvolve combine model proposals with execution feedback and search or evolution (Zhang et al., 2026; Novikov et al., 2025; Liao et al., 2025). These LLM-guided methods keep the proposing models fixed.

Learning optimizations and kernels. Cummins et al. and LLM Compiler train models for compiler optimization (Cummins et al., 2023; 2024); Compiler-R1 and ECCO incorporate performance-rewarded training (Pan et al., 2025; 2026). TLM and Mehmood et al. learn schedule generation (Zhai et al., 2024; Mehmood et al., 2026). Kevin, AutoTriton, Dr. Kernel, and CUDA Agent train kernel generators with execution feedback (Baronio et al., 2025; Li et al., 2025; Liu et al., 2026; Dai et al., 2026), while TritonRL adds safeguards against reward hacking (Woo et al., 2025).

Representations and our approach. Learned edit representations transfer program changes (Yin et al., 2019), while CAKE exposes typed schedules and evolves its compiler harness (Ye et al., 2026). Our policy learns complete construction and refinement schedules through search-derived supervision and execution-grounded RL. Our typed edit language reconstructs exact compiler traces, with explanations grounded in compiler-derived facts. Replay and compilation transform the existing computation into a kernel, which undergoes numerical validation.

5 Conclusion

We introduced How to Train Your Little Dragon for training a small language model to construct and refine complete GPU kernel optimization scripts. Our compact representation enables exact reconstruction of compiler traces. SFT teaches construction and refinement tasks from well-performing verified compiler search examples; RL improves schedule choices through correctness and latency feedback. The trained 8B model achieves Correct@1 of 91.27% for construction and 92.01% for refinement, outperforming six baselines across all reported correctness and performance-success metrics. These results establish compiler search experience and execution feedback as effective training signals for a policy that coordinates compiler optimization scripts in one complete proposal.

AI USE STATEMENT

In this work, we used generative AI tools for polishing writing, such as refining the language and improving readability. We also used AI tools for generating synthetic datasets. We used the opensource model Kimi-K3 to help with generating our training dataset. We have not used generative AI tools for retrieval and discovery, for research ideation or execution, for drafting sections of the paper, or for proving mathematical claims. We take responsibility for the final content of this work, including text polished and training dataset produced with the aid of generative AI.

REFERENCES

Anthropic. Claude Opus 5 system card. Technical report, Anthropic, July 2026. URL https://anthropic.com/claude-opus-5-system-card.

Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. Kevin: Multi-turn RL for generating CUDA kernels. *arXiv preprint arXiv:2507.11948*, 2025. URL https://arxiv.org/abs/2507.11948.

ByteDance Seed, Yuyu Zhang, Jing Su, Yifan Sun, Chenguang Xi, Xia Xiao, Shen Zheng, Anxiang Zhang, Kaibo Liu, Daoguang Zan, et al. Seed-Coder: Let the code model curate data for itself. *arXiv preprint arXiv:2506.03524*, 2025. URL https://arxiv.org/abs/2506.03524.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. URL https://arxiv.org/abs/2107.03374.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 578–594, 2018a. URL https://arxiv.org/abs/1802.04799.

Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018b. URL https://proceedings.neurips.cc/paper/2018/hash/8b5700012be65c9da25f49408d959ca0-Abstract.html.

Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Kim Hazelwood, Gabriel Synnaeve, and Hugh Leather. Large language models for compiler optimization. *arXiv preprint arXiv:2309.07062*, 2023. URL https://arxiv.org/abs/2309.07062.

Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Rozière, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. Meta large language model compiler: Foundation models of compiler optimization. *arXiv preprint arXiv:2407.02524*, 2024. URL https://arxiv.org/abs/2407.02524.

Weinan Dai, Hanlin Wu, Qiying Yu, Huan-ang Gao, Jiahao Li, Chengquan Jiang, Weiqiang Lou, Yufan Song, Hongli Yu, Jiaze Chen, Wei-Ying Ma, Ya-Qin Zhang, Jingjing Liu, Mingxuan Wang, Xin Liu, and Hao Zhou. CUDA Agent: Large-scale agentic RL for high-performance CUDA kernel generation. *arXiv preprint arXiv:2602.24286*, 2026. URL https://arxiv.org/abs/2602.24286.

Gyuhak Kim, Sumiran Singh Thakur, Su Min Park, Wei Wei, and Yujia Bao. SFT-GO: Supervised fine-tuning with group optimization for large language models. *arXiv preprint arXiv:2506.15021*, 2025. URL https://arxiv.org/abs/2506.15021.

Kimi Team. Kimi K3: Open frontier intelligence. Technical report, Moonshot AI, 2026. URL https://github.com/MoonshotAI/Kimi-K3/blob/main/k3_tech_report.pdf.

Shangzhan Li, Zefan Wang, Ye He, Yuxuan Li, Qi Shi, Jianling Li, Yonggang Hu, Wanxiang Che, Xu Han, Zhiyuan Liu, and Maosong Sun. AutoTriton: Automatic triton programming with reinforcement learning in LLMs. *arXiv preprint arXiv:2507.05687*, 2025. URL https://arxiv.org/abs/2507.05687.

Xuan Li, Yining Wang, Yuchen Liu, Guanjun Liu, Delai Qiu, Shengping Liu, Jiaen Liang, Wei Huang, Jun Yu, and Junnan Zhu. LaTER: Efficient test-time reasoning via latent exploration and explicit verification. *arXiv preprint arXiv:2605.07315*, 2026. URL https://arxiv.org/abs/2605.07315.

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, Liyuan Li, 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. *arXiv preprint arXiv:2512.23236*, 2025. URL https://arxiv.org/abs/2512.23236.

Wei Liu, Jiawei Xu, Yingru Li, Longtao Zheng, Tianjian Li, Qian Liu, and Junxian He. Dr. kernel: Reinforcement learning done right for triton kernel generations. *arXiv preprint arXiv:2602.05885*, 2026. URL https://arxiv.org/abs/2602.05885.

Sajid Mehmood, Aqleema Arooj, Ahmad Sami Al-Shamayleh, Samera Batool, Inayat Ur-Rehman, Muhammad Dilshad Sabir, Sabeen Masood, and Adnan Akhunzada. Tensor language model enables generative scheduling for efficient tensor compilation. *Scientific Reports*, 16:15379, 2026. doi: 10.1038/s41598-026-41392-8. URL https://www.nature.com/articles/s41598-026-41392-8.

Alexander Novikov, Ngân Vũ, 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. *arXiv preprint arXiv:2506.13131*, 2025. URL https://arxiv.org/abs/2506.13131.

OpenAI. GPT-6 Astra system card. Technical report, OpenAI, September 2026. URL https://deploymentsafety.openai.com/gpt-6-astra.

Haolin Pan, Hongyu Lin, Haoran Luo, Yang Liu, Kaichun Yao, Libo Zhang, Mingjie Xing, and Yanjun Wu. Compiler-R1: Towards agentic compiler auto-tuning with reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 38, 2025. URL https://arxiv.org/abs/2506.15701.

Haolin Pan, Lianghong Huang, Jinyuan Dong, Mingjie Xing, and Yanjun Wu. ECCO: Evidence-driven causal reasoning for compiler optimization. *arXiv preprint arXiv:2602.00087*, 2026. URL https://arxiv.org/abs/2602.00087.

Qwen Team. Qwen3.8-27B. Hugging Face model card, 2026. URL https://huggingface.co/Qwen/Qwen3.8-27B. Accessed September 20, 2026.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017. URL https://arxiv.org/abs/1707.06347.

Junru Shao, Xiyou Zhou, Siyuan Feng, Bohan Hou, Ruihang Lai, Hongyi Jin, Wuwei Lin, Masahiro Masuda, Cody Hao Yu, and Tianqi Chen. Tensor program optimization with probabilistic programs. In *Advances in Neural Information Processing Systems*, volume 35, 2022. URL https://arxiv.org/abs/2205.13603.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024. URL https://arxiv.org/abs/2402.03300.

Annabelle Sujun Tang, Christopher Priebe, Rohan Mahapatra, Lianhui Qin, and Hadi Esmaeilzadeh. REASONING COMPILER: LLM-guided optimizations for efficient model serving. In *Advances in Neural Information Processing Systems*, 2025. URL https://arxiv.org/abs/2506.01374v5.

Annabelle Sujun Tang, Christopher Priebe, Lianhui Qin, and Hadi Esmaeilzadeh. LiteCoOp: Lightweight multi-LLM shared-tree reasoning for model-serving compiler optimizations. *arXiv preprint arXiv:2602.01935*, 2026. URL https://arxiv.org/abs/2602.01935v2.

Jiin Woo, Shaowei Zhu, Allen Nie, Zhen Jia, Yida Wang, and Youngsuk Park. TritonRL: Training LLMs to think and code triton without cheating. *arXiv preprint arXiv:2510.17891*, 2025. URL https://arxiv.org/abs/2510.17891.

Zihao Ye, Yingyi Huang, Hongyi Jin, Bohan Hou, Junru Shao, Zhongming Yu, Jinqi Chen, Meghan Cowan, Shiyi Cao, Shanli Xing, Hanfeng Chen, Vinod Grover, Tianqi Chen, and Luis Ceze. CAKE: Compiler-agent co-design for frontier kernel evolution. *arXiv preprint arXiv:2608.12629*, 2026. URL https://arxiv.org/abs/2608.12629.

Pengcheng Yin, Graham Neubig, Miltiadis Allamanis, Marc Brockschmidt, and Alexander L. Gaunt. Learning to represent edits. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=BJl6AjC5F7.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, et al. DAPO: An open-source LLM reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025. URL https://arxiv.org/abs/2503.14476.

Z.ai. GLM-5.3. Hugging Face model card, 2026. URL https://huggingface.co/zai-org/GLM-5.3. Accessed September 20, 2026.

Yi Zhai, Sijia Yang, Keyu Pan, Renwei Zhang, Shuo Liu, Chao Liu, Zichun Ye, Jianmin Ji, Jie Zhao, Yu Zhang, and Yanyong Zhang. Enabling tensor language model to assist in generating high-performance tensor programs for deep learning. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 289–305. USENIX Association, 2024. URL https://www.usenix.org/conference/osdi24/presentation/zhai.

Genghan Zhang, Shaowei Zhu, Anjiang Wei, Zhenyu Song, Allen Nie, Zhen Jia, Nandita Vijaykumar, Yida Wang, and Kunle Olukotun. AccelOpt: A self-improving LLM agentic system for AI accelerator kernel optimization. In *Proceedings of Machine Learning and Systems*, volume 8, pp. 541–568, 2026. URL https://proceedings.mlsys.org/paper_files/paper/2026/file/0f84265589057 46fc38da5e335700aec-Paper-Conference.pdf.

Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 863–879, 2020. URL https://arxiv.org/abs/2006.06762.

A REPRESENTATION AND TRAINING-LABEL EXAMPLE

A.1 A CONCRETE POLICY INPUT

The construction input $x = (d, M_0, F(M_0), h)$ below comes from the projection training example used in Figure 2. It computes $Y = \mathrm{cast}_{\mathrm{FP16}}(XW^\top)$. The task description and feature record are abbreviated for readability. TensorIR identifiers are shortened, with X, W, C, and Y denoting the buffers.

Task description d. "Optimize the extracted TensorIR task `fused_NT_matmul_cast`: an NT projection with 112 rows, input width 3456, and output width 4304, for lower latency on NVIDIA H100. The generated schedule must preserve numerical correctness."

Initial module M_0. The unscheduled module contains a matrix-product reduction followed by an output cast. Its serial loops specify the computation before tiling, thread binding, or tensorization:

```
from tvm.script import ir as I, tirx as T

@I.ir_module
class Module:
    @T.prim_func(s_tir=True)
    def main(X: T.Buffer((T.int64(112), T.int64(3456)), "float16"),
             W: T.Buffer((T.int64(4304), T.int64(3456)), "float16"),
             Y: T.Buffer((T.int64(112), T.int64(4304)), "float16")):
        T.func_attr({"tirx.noalias": True})
        C = T.sblock_alloc_buffer((T.int64(112), T.int64(4304)), "float32")
        for i, j, k in T.grid(T.int64(112), T.int64(4304), T.int64(3456)):
            with T.sblock("NT_matmul"):
                vi, vj, vk = T.axis.remap("SSR", [i, j, k])
                T.reads(X[vi, vk], W[vj, vk])
                T.writes(C[vi, vj])
                with T.init():
                    C[vi, vj] = T.float32(0.0)
                C[vi, vj] = C[vi, vj] + (
                    T.Cast("float32", X[vi, vk]) * T.Cast("float32", W[vj, vk]))
        for i, j in T.grid(T.int64(112), T.int64(4304)):
            with T.sblock("compute"):
                vi, vj = T.axis.remap("SS", [i, j])
                T.reads(C[vi, vj])
                T.writes(Y[vi, vj])
                Y[vi, vj] = T.Cast("float16", C[vi, vj])
```

Structural features $F(M_0)$. Deterministic AST analysis records one function, five serial loops, and no thread bindings. A subset of the actual feature fields is:

```
{"primfunc_count": 1, "loop_kind_counts": {"serial": 5},
 "thread_binding_counts": {}, "storage_scope_counts": {"global": 4}}
```

The same record lists the buffers' shapes and dtypes, the reduction block's two spatial axes of extents 112 and 4304 and reduction axis of extent 3456 (`SSR`), and the cast block's two spatial axes (`SS`). Its read/write regions show that the reduction reads X and W and writes the intermediate C, which the cast reads to produce Y. These facts are extracted from M_0 before a schedule is proposed.

Compiler and target environment h. The recorded environment uses TVM `0.26.dev0`, and an NVIDIA H100 80GB HBM3 CUDA target with architecture `sm_90`. The target configuration specifies a warp size of 32, at most 1024 threads per block, and a shared-memory-per-block limit of 49,152 bytes.

A.2 THE TWO LAYERS OF AN OPTIMIZATION GRAPH

The executable payload and the explanation layer have different contracts. The payload must reconstruct one complete pinned TVM trace, including its sampling choices and postprocessing. Its validity is tested by the compiler and empirical execution. The explanation layer identifies evidence, proposes reasons for choices, and links those reasons to edits. During label generation, teacher

explanations are checked against facts from the supplied input and the actual payload; these checks do not provide a formal proof of performance causality.

The serialization uses an XML graph containing JSON trace fragments. The exact order of `trace_sequence` nodes and their instruction lists determines execution. The principal instruction fields are `k` (kind), `i` (inputs), `a` (attributes), `o` (output type signature), and a sampled decision or its lossless derived-factor encoding. A reference such as `t22.0.2` denotes output 2 of local step 0 in edit group `t22`. Type signatures distinguish block, loop, and expression values. These local coordinates avoid asking the model to maintain a global output counter, but still require valid backward references.

Losslessness scope. Equation 2 concerns the canonical TVM trace representation under the recorded codec/compiler version. It does not imply arbitrary historical Python whitespace reproduction, equivalence between compiler versions, or universal support for every possible TVM object. Accepted dataset traces are round-trip checked. Unsupported constructs must fail validation rather than be silently approximated. The lowering layer retains every instruction even when the explanatory description groups several instructions into one semantic edit.

B RL POLICY OPTIMIZATION

For eight sampled completions, the trainer forms group-standardized advantages $A_i = (R_i - \overline{R})/(s_R + \epsilon)$ and uses the PPO-style clipped likelihood-ratio objective (Schulman et al., 2017) of the form

$$J(\theta) = \mathbb{E}\left[\frac{1}{8}\sum_{i=1}^{8}\frac{1}{|y_i|}\sum_{t}\min(r_{i,t}(\theta)A_i, \text{clip}(r_{i,t}(\theta), 1-0.2, 1+0.2)A_i)\right]. \tag{8}$$

$r_{i,t}$ is the new-policy to rollout-policy token probability ratio. There is no KL term. Dynamic sampling drops groups with zero reward variance, following the filtering idea in DAPO (Yu et al., 2025), and obtains additional groups from an oversampled pool (160 prompts for a retained batch of 128). This makes the retained training distribution policy-dependent. The 60,800 target rows are selected from an SFT-screened pool with a 60:40 construction/refinement mixture, so training performance is not a random estimate of the entire workload universe.

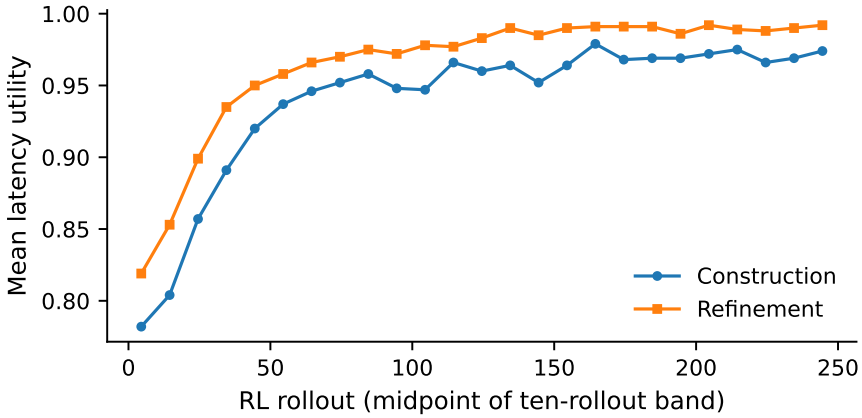

Figure 5: **RL training utility.** Ten-rollout means, plotted at band midpoints. The series is measured on screened, dynamically retained training samples, not the sealed set.

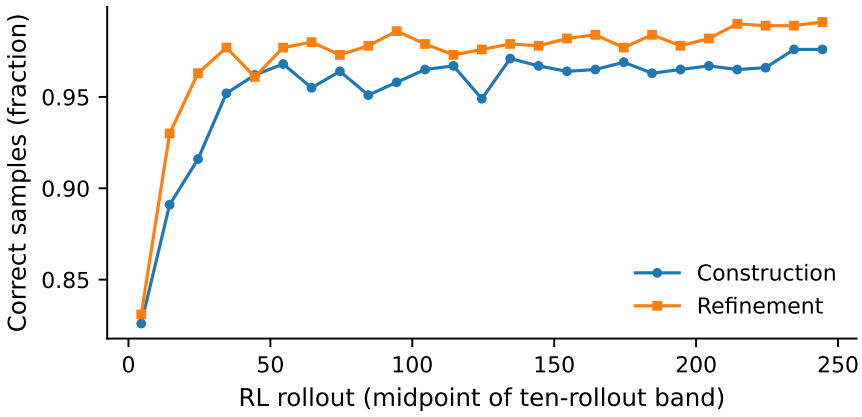

Figure 6: **RL training correctness.** Ten-rollout sample-correctness summaries.