LESS IS MORE: LEARNING WHAT TO PROFILE FOR GPU KERNEL OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models have recently shown strong potential for automatically optimizing GPU kernels, yet their effectiveness depends critically on the hardware information available to the kernel optimization model. Existing systems typically rely on predefined profiling strategies, leaving *what to profile* largely outside the optimization process. We show that effective profiling is not simply about collecting more information: increasing the number of profiling metrics does not monotonically improve kernel optimization, while the most useful profiling information varies across kernels. Our theoretical analysis further shows that, under such heterogeneous profiling utility, kernel-conditioned selection can outperform any single fixed profiling configuration. Motivated by this observation, we propose *AutoProfiler*, a feedback-driven framework for learning a deterministic profiling tree from optimization feedback. The tree is iteratively refined through REWRITE and SPLIT operations, with an LLM generating candidate modifications and empirical evaluation determining their acceptance. Once constructed, the tree quickly gathers the required profiling information for any kernel, allowing the LLM rewriter to be invoked only once. Experiments on KernelBench show that our approach improves kernel optimization over fixed profiling configurations under constrained profiling budgets. These results suggest that hardware profiling should evolve from static metric collection toward profiling reasoning, making profiling itself a new dimension for improving kernel optimization.

1 INTRODUCTION

LLM-based systems have recently shown strong potential for automatically rewriting GPU kernels to improve execution efficiency Ouyang et al. (2025); Zhang et al. (2025); Gai et al. (2026a). To better guide the rewriting process, recent systems increasingly incorporate hardware profiling to expose runtime bottlenecks and provide actionable optimization feedback Zhang et al. (2025); Gai et al. (2026a). However, existing approaches typically collect a predefined set of profiling signals before optimization. While simple and reusable, this design overlooks two fundamental properties of hardware profiling: profiling information is costly to acquire, and its utility is inherently kernel-dependent.

This raises a fundamental question: *What profiling information is actually needed for effective kernel optimization?* We investigate this question from two complementary perspectives. First, more profiling is not necessarily better. Empirically, increasing the number of profiling metrics does not monotonically improve kernel optimization, suggesting that more profiling information does not necessarily translate into better optimization. Second, the useful information is kernel-dependent. Our theoretical analysis shows that when profiling utility varies across kernels, kernel-conditioned selection can outperform any single fixed configuration, which is further supported by our empirical oracle analysis. Together, these findings suggest that effective profiling is not about collecting more information, but about identifying the right information for each kernel.

To this end, we propose AutoProfiler, which learns a kernel-conditioned profiling policy that adaptively acquires profiling metrics before invoking the kernel rewriter. We formulate profiling as sequential information acquisition and represent the policy with a deterministic profiling tree, which selects subsequent measurements based on previously observed profiling results. The tree is refined through optimization feedback: an LLM proposes structural REWRITE to deepen the tree by col-

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.

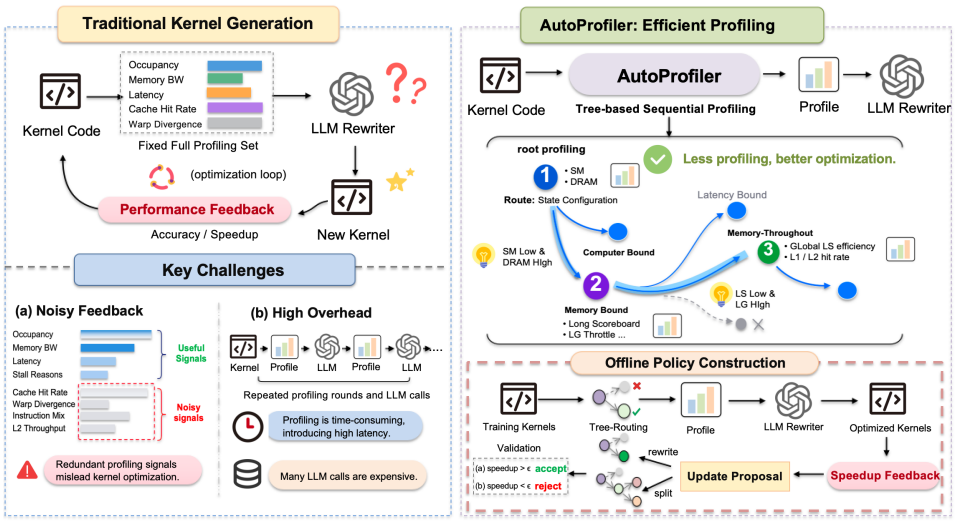


Figure 1: Overview and motivation of our profiling reasoning framework. **Left:** Traditional kernel optimization and its key challenges, where redundant profiling signals introduce noisy feedback (a) and repeated profiling rounds incur high latency and LLM overhead (b). **Right:** Our AutoProfiler method utilizes tree-based sequential profiling to selectively acquire informative signals, achieving effective optimization with reduced profiling overhead.

lecting more profiling information, and SPLIT to widen the tree for kernels with different profiling needs, while empirical paired evaluation determines whether a proposed modification is accepted. The LLM is used only during policy construction, leaving inference-time routing fully deterministic.

Experiments on KernelBench show that AutoProfiler improves kernel optimization over fixed profiling configurations under a constrained profiling budget, while recovering a substantial portion of the gain available from kernel-conditioned selection. Interestingly, the learned profiling policy remains effective across iterative optimization rounds, enabling continued gains without repeated LLM-based profiling decisions. Together, these results point to a promising direction for self-improving kernel generation: using more targeted profiling to provide effective feedback at lower cost, thereby supporting continued improvement through iterative optimization.

2 Preliminary

Notations. Let $k \in \mathcal{K}$ denote a GPU kernel sampled from a kernel corpus, and let $\mathcal{M}$ denote the set of available profiling metrics. Given a subset of metrics $m \subseteq \mathcal{M}$, a profiler P produces the corresponding profiling information $I = P(k, m)$. A kernel rewriter g then uses the original kernel and profiling information to generate an optimized kernel: $\hat{k} = g(k, I) = g\big(k, P(k, m)\big)$. Throughout this work, the profiler P, the rewriter g, and the target hardware are fixed.

Problem setup. We formulate the GPU kernel optimization as a profiling-conditioned generation problem: given an input kernel, the optimized kernel is generated conditioned on both the kernel itself and the acquired hardware profiling information. Existing approaches typically use a fixed profiling configuration for all kernels. Given a predefined metric set $m_0 \subseteq \mathcal{M}$, the optimized kernel is generated as $\hat{k} = g\big(k, P(k, m_0)\big)$, where the same set of profiling metrics is collected regardless of the kernel being optimized. In contrast, we allow the profiling configuration to vary across kernels. Let $\phi(k) \subseteq \mathcal{M}$ denote the profiling metrics selected for kernel k. The optimized kernel is then generated as

$$\hat{k} = g\big(k, P(k, \phi(k))\big). \tag{1}$$

This formulation turns profiling into a sequential reasoning process over hardware signals: based on the profiling information observed so far, the policy determines what additional information is most valuable to acquire next for kernel optimization.

Our goal. We aim to learn a profiling policy ϕ that determines what profiling information to acquire so as to improve kernel optimization. We measure the utility of the acquired profiling information by the performance of the resulting kernel. Specifically, for a kernel k and profiling metrics m, we define the resulting speedup as $s(k, m) = \frac{\text{lat}(k)}{\text{lat}(\tilde{k})}$, where $\text{lat}(\cdot)$ denotes the execution latency on the target hardware. The corresponding optimization reward is $r(k, m) = \max\{s(k, m) - 1, 0\}$, which assigns positive reward only when the acquired profiling information leads to a rewrite that improves upon the original kernel. Our objective is therefore to learn a profiling policy that maximizes the expected optimization reward:

$$\arg\max_{\phi} \mathbb{E}_{k\sim\mathcal{K}} \left[r\big(k, \phi(k)\big) \right], \tag{2}$$

subject to a profiling budget that limits the amount of hardware information acquired for each kernel, where $\mathcal{K}$ denotes the distribution over kernels under consideration. The key challenge is therefore to exploit kernel-dependent profiling utility while avoiding unnecessary profiling cost.

3 Static vs. Dynamic Metric Selection: Theoretical and Empirical Analysis

Theoretical Analysis. We first characterize the fundamental advantage of dynamic profiling over static metric selection. Specifically, we show that dynamic selection can never perform worse than the best fixed profiling configuration, and that a strict advantage arises when different configurations are effective for different kernels. Consider a finite family of candidate profiling configurations $\mathcal{C} = \{m_1, m_2, \ldots, m_K\}$ and a kernel distribution $\mathcal{D}$. Let $u(k, m)$ denote the kernel optimization utility (*e.g.,* performance speedup) for kernel k under configuration m. The expected utility of the optimal static configuration is defined as $U_{\text{static}} = \max_{m\in\mathcal{C}} \mathbb{E}_{k\sim\mathcal{D}}[u(k, m)]$, whereas an oracle dynamic policy that adaptively selects the most effective configuration per kernel achieves $U_{\text{dynamic}} = \mathbb{E}_{k\sim\mathcal{D}}[\max_{m\in\mathcal{C}} u(k, m)]$.

Proposition 1. Dynamic selection dominates static selection. For any finite candidate family $\mathcal{C}$,

$$U_{\text{dynamic}} \geq U_{\text{static}}. \tag{3}$$

Moreover, equality holds if and only if there exists a configuration $m_j \in \mathcal{C}$ whose success region covers the union of all candidate success regions almost surely under $\mathcal{D}$. Otherwise, $U_{\text{dynamic}} > U_{\text{static}}$. This result shows that the value of dynamic profiling is determined by the heterogeneity of profiling utility across kernels: when no single configuration covers the success regions of the others, adapting the profiling configuration to each kernel yields a strict advantage.

Empirical Validation. We next examine whether the heterogeneity underlying Proposition 1 occurs in practice. We conduct an oracle-style evaluation on 128 KernelBench kernels using four roofline-based profiling configurations. For each kernel, we evaluate the same frozen optimizer under each configuration and select the configuration yielding the highest speedup. Detailed experimental settings are provided in Appendix A.2. We find that the per-kernel oracle achieves a geomean speedup of $1.429\times$, compared with $1.332\times$ for the best fixed configuration, corresponding to a 7.3% relative gain. Moreover, the oracle selects different configurations across kernels, with the best configuration varying substantially among the four candidates.

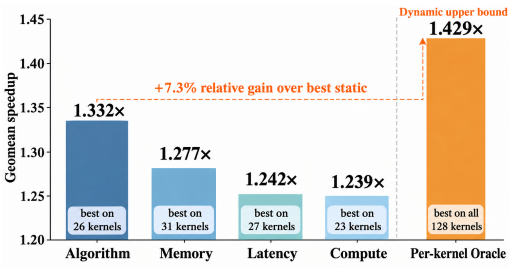


Figure 2: Per-kernel oracle evaluation on 128 KernelBench kernels, showing substantial heterogeneity in the optimal profiling configuration and a 7.3% relative gain over the best fixed configuration.

Key Findings on Profiling Utility

- **Profiling utility is heterogeneous across kernels.**
  Different kernels benefit from different profiling signals, and no single configuration is consistently optimal.
- **Kernel-specific profiling provides substantial optimization headroom.**
  The oracle gap shows that effective optimization depends on selecting useful profiling information for each kernel rather than collecting a universally strong metric set.

This result provides empirical support for the heterogeneity condition in Proposition 1 and indicates non-trivial room for kernel-conditioned profiling. The oracle serves only as an analysis reference; it requires evaluating all candidate configurations for each kernel and is unavailable at inference time. This motivates the central problem addressed next: learning which profiling configuration to use for a given kernel without exhaustively evaluating all candidates.

4 METHODS

To optimize each kernel with the most informative profiling feedback, we aim to identify what information is most useful at each stage of optimization. In this section, we present AutoProfiler, a feedback-driven framework for learning kernel-conditioned hardware profiling policies. AutoProfiler formulates hardware profiling as a sequential decision process and instantiates the learned policy with a deterministic profiling tree. Our methodology is structured as follows: we first introduce the Tree-based Profiling Policy formulation (Section 4.1), detail its Feedback-driven Policy Construction via LLM-based structural updates and describe its Deterministic Inference-time Routing pipeline (Section 4.2).

4.1 TREE-BASED PROFILING POLICY

AutoProfiler model profiling as a sequential decision process and implement the resulting policy with a deterministic tree. We first formalize how profiling information is acquired sequentially, then describe how this policy is represented by the tree, and finally explain how the learned tree guides profiling for an input kernel.

Sequential Profiling. To formalize this process, we formulate profiling as a sequential policy that determines what hardware information to collect next based on the profiling observations obtained so far. For a kernel k, at profiling step t, the current state is denoted by

$$s_t = (M_t, I_t),$$

where $M_t \subseteq \mathcal{M}$ is the set of metrics collected so far and $I_t = P(k, M_t)$ is the corresponding profiling information. The policy then selects an additional metric group

$$G_t = \pi_T(s_t), \qquad G_t \subseteq \mathcal{M} \setminus M_t,$$

which leads to the next profiling state

$$s_{t+1} = (M_{t+1}, I_{t+1}), \qquad M_{t+1} = M_t \cup G_t, \quad I_{t+1} = P(k, M_{t+1}).$$

Equivalently, the profiling process can be viewed as

$$s_t \xrightarrow{\pi_T} G_t \xrightarrow{\text{collect } G_t} s_{t+1}.$$

Profiling terminates when the policy determines that no additional information is required. In this way, the metrics collected for a kernel depend on its observed hardware behavior rather than being fixed in advance.

Tree Representation. To instantiate this sequential policy, we represent π_T as a rooted tree $T = (V, E, v_0)$. Specifically, the root node v_0 captures an initial profiling state using two fundamental microarchitectural signals: SM (Compute) utilization and DRAM (Memory) throughput. Guided by the roofline model, these two metrics classify any input kernel into one of four canonical

bottleneck regimes: *Latency-bound*, *Memory-bound*, *Compute-bound*, or *Algorithm-bound*. This taxonomy forms the four initial outgoing edges from v_0, establishing a coarse-grained routing foundation before fine-grained metric selection. From this initial routing, the tree progressively grows into two types of nodes with distinct roles:

- **Internal node:** This specifies a routing condition over the profiling information collected so far, with outgoing edges corresponding to different outcomes of the condition.
- **Leaf:** This represents a terminal profiling state and determines the final metric configuration for the kernels reaching it.

Tree-Guided Profiling. Given a kernel k, the profiling tree sequentially evaluates its routing conditions and collects the required metrics until reaching a leaf. This produces a kernel-specific metric set $E_T(k)$ and the corresponding profiling information

$$I_T(k) = P(k, E_T(k)) .$$

Importantly, all profiling decisions are completed by the deterministic tree without LLM-based routing. Once the complete profiling information has been collected, it is provided to the LLM rewriter in a single inference:

$$\hat{k}_T = g(k, I_T(k)) .$$

Thus, the tree determines what hardware evidence to acquire, while the LLM consumes the complete evidence once to generate the final optimized kernel.

4.2 Feedback-Driven Policy Construction

We next turn to how the profiling tree itself is constructed. The construction proceeds iteratively through three steps: collecting leaf-level optimization feedback, proposing local Rewrite or Split operations, and retaining refinements that improve kernel optimization.

4.2.1 Leaf-Level Attribution and Structural Updates

Without loss of generality, we consider the current tree T with root v_0 and a leaf v_i. We route N training kernels $\{k_j\}_{j=1}^{N}$ through T, among which N_i kernels eventually reach v_i. Following the sequential profiling process defined in Section 4.1, each such kernel k_j arrives at v_i with a terminal profiling state

$$s_{t_i}^{(j)} = \left(M_{t_i}^{(j)}, I_{t_i}^{(j)} \right) ,$$

where $M_{t_i}^{(j)}$ is the set of metrics collected along its routing path and $I_{t_i}^{(j)} = P(k_j, M_{t_i}^{(j)})$ is the corresponding profiling information. After the collected information is provided to the fixed LLM rewriter, executing the resulting kernel on the target hardware yields the optimization reward $r_T(k_j)$.

Therefore, for the leaf v_i we obtain two forms of local evidence: their terminal profiling observations $I_{t_i}^{(j)}$, which characterize the observed hardware behavior, and their optimization rewards $r_T(k_j)$, which reflect the resulting optimization performance. Based on this local evidence, we consider two complementary forms of tree growth : Rewrite enriches the profiling information collected at v_i, while Split further partitions the kernels reaching v_i into different profiling regimes.

Rewrite. One possible limitation of the current leaf is that its profiling configuration does not expose sufficient information for effective optimization. We therefore consider augmenting its metric configuration with an additional metric group G:

$$M_v' = M_v \cup G, \qquad G \subseteq \mathcal{M} \setminus M_v.$$

Rewrite preserves the routing structure while enriching the information available to all kernels reaching v.

Split. Another possibility is that kernels reaching the same leaf benefit from different profiling information. We therefore consider partitioning the leaf according to an already-collected profiling feature x and threshold θ:

$$v \to \begin{cases} v_L, & x \le \theta, \\ v_R, & x > \theta. \end{cases}$$

The resulting child leaves can adopt different profiling configurations, allowing the policy to specialize subsequent profiling decisions according to the observed hardware behavior.

For both operators, an LLM uses the local feedback to propose candidate refinements: additional metric groups for REWRITE, or feature–threshold pairs for SPLIT. The LLM only generates candidate modifications; whether a candidate is incorporated into the tree is determined by empirical evaluation in the subsequent stage.

4.2.2 CONSERVATIVE EMPIRICAL ACCEPTANCE

A candidate modification is accepted only if it demonstrates sufficient downstream improvement over the incumbent tree. For a candidate tree T', we evaluate T' and the incumbent T on the same set of training kernels and compute their paired reward difference:

$$\frac{1}{N}\sum_{k=1}^{N}\left[r_{T'}(k) - r_T(k)\right],$$

where $r_{T'}(k)$ and $r_T(k)$ denote the downstream rewards obtained by kernel (k) under the candidate and incumbent trees, respectively. The candidate is accepted only when

$$\widehat{\Delta}(T', T) > \epsilon,$$

where ϵ is a conservative acceptance margin.

Paired evaluation controls for kernel-level variation by evaluating the candidate and incumbent trees on the same kernels, isolating the effect of the structural modification. When the rewriter involves stochastic sampling, the same sampling seed is used for the paired evaluations. We apply the operators conservatively. REWRITE is considered first as the minimal intervention for potential information insufficiency at the current leaf. If no REWRITE candidate satisfies the acceptance criterion, SPLIT is considered to capture potential heterogeneity within that leaf. A rejected modification leaves the incumbent tree unchanged. Together with the profiling budget (B), this controls the complexity of the learned tree and the amount of profiling information collected for each kernel.

Once construction terminates, the resulting tree is fixed and used for inference. The LLM is involved only during offline policy construction; inference-time routing follows the learned profiling conditions deterministically and requires no LLM.

5 EXPERIMENTS

5.1 EXPERIMENTAL SETUP

Datasets. We use separate datasets for profiling-policy construction and evaluation to avoid overlap between training and test kernels. The profiling tree is constructed solely on the CUDA-Agent-Ops-6K corpus without exposure to KernelBench. As a result, KernelBench serves as an out-of-distribution benchmark to evaluate the policy's transferability to unseen kernels.

For training, we use CUDA-Agent-Ops-6K (Dai et al., 2026), a curated dataset of 6,000 synthesized operator-level tasks released for CUDA kernel generation and optimization. The dataset is constructed by collecting reference operators from torch and transformers, synthesizing multi-operator tasks with LLMs, and filtering them based on execution correctness, determinism, runtime, and similarity to KernelBench test cases. This provides a diverse training corpus while reducing contamination with the evaluation benchmark.

For evaluation, we use KernelBench (Ouyang et al., 2025), a widely used benchmark for LLM-based GPU kernel optimization. It contains three levels of increasing complexity: Level 1 (100 problems) focuses on individual operators such as matrix multiplication and convolution; Level 2 (100 problems) targets fused operators such as Conv+ReLU+Bias; and Level 3 (50 problems) consists of complete models such as MobileNet, VGG, and MiniGPT. These levels cover a broad range of operator complexity and problem scale.

Baselines. We compare AutoProfiler with four training-free agentic kernel-optimization methods: CudaForge (Zhang et al., 2025), which iteratively compiles, benchmarks, and repairs generated ker-

Table 1: Main performance comparison on KernelBench. "Speedup" denotes the geometric mean speedup over correctly solved problems, while "Accuracy" denotes the proportion of problems solved correctly. "LLM Calls/Task" indicates the average number of LLM inferences required per optimization task.

Method	Speedup (↑)				Avg. Accuracy (%)	LLM Calls/Task
	Level 1	Level 2	Level 3	Avg.		
CudaForge	1.567×	**1.808**×	1.099×	1.570×	78.4	2
KernelMem	1.233×	1.351×	1.091×	1.252×	82.0	2
KernelAgent	1.361×	1.306×	1.044×	1.276×	84.8	4
AutoKernel	1.058×	1.182×	0.997×	1.095×	84.4	1
AutoProfiler	**1.810**×	1.775×	**1.352**×	**1.704**×	**87.2**	**1**

nels; KernelMem (Sun et al., 2026), which incorporates previous optimization experiences through memory-augmented prompting; KernelAgent (Cheng et al., 2026), which searches for improved kernels using correctness and performance feedback; and AutoKernel (Jaber & Jaber, 2026), which progressively refines kernels through single-edit modifications with rollback. These methods cover different inference-time optimization strategies, providing a diverse set of baselines for evaluating AutoProfiler.

Evaluation Metrics. Following the prior kernel optimization works Zhang et al. (2026); Ran et al. (2025), we evaluate model performance using execution accuracy to measure correctness and geometric-mean speedup to measure efficiency. Execution accuracy measures the proportion of problems where the generated kernel is both compilable and numerically correct against the PyTorch reference. Geometric-mean speedup is computed relative to the reference kernel across all problems, providing a scale-robust aggregate metric. Unlike the arithmetic mean, which is prone to severe distortion from extreme outliers, the geometric mean ensures that no single task disproportionately dominates the aggregate score, yielding a more balanced and representative evaluation (Fleming & Wallace, 1986).

Implementations. We employ GPT-5.6-Sol as the backbone kernel rewriter across AutoProfiler and all baseline methods. To ensure a fair comparison under equal inference budgets, each method is restricted to a single optimization pass (round $= 1$). For hardware profiling, baselines adopt their original static NCU metric configurations, whereas AutoProfiler dynamically selects profiling metrics guided by its learned policy. All methods share an identical NCU backend on four NVIDIA RTX 4090D GPUs (Ada Lovelace, 24GB). Specifically, the AutoProfiler rewriter targets Triton kernel generation. To isolate performance gains driven by profiling policy from baseline framework translation artifacts, each original PyTorch operator is first translated into a correctness-verified Triton seed prioritizing numerical fidelity over performance.

5.2 Main Results

Table 1 summarizes the performance across different methods on KernelBench. AutoProfiler achieves the highest average speedup of 1.704×, outperforming the strongest baseline, CudaForge (1.570×) by 8.5%. Notably, the performance advantage of AutoProfiler is most pronounced on the challenging Level 3 task, where it elevates the speedup from 1.099× of CudaForge to 1.352×. This substantial gain demonstrates that kernel-conditioned profile selection scales effectively to complex optimization spaces, going beyond simple operator-level enhancements.

In terms of correctness, AutoProfiler attains the highest execution accuracy at 87.2%, outperforming other baselines by 2.4–8.8 percentage points. Crucially, these efficiency and accuracy gains are achieved with minimal compute overhead: AutoProfiler requires only one single LLM call per task, whereas CudaForge and KernelMem consume 2 calls, and KernelAgent requires 4 calls. These results substantiate that profiling-guided hardware feedback serves as an efficient, lightweight prior for LLM-driven kernel optimization, delivering superior optimization quality while markedly reducing generation costs.

Table 2: Effect of profiling information on downstream kernel optimization.

Condition	Speedup (↑)
Blind	1.197
Random	1.215
Static	1.314
AutoProfiler	**1.379**
Oracle	1.573

Table 3: Impact of profiling information quantity on downstream kernel optimization.

# Metrics	Regime	Speedup (↑)
2	Insufficiency	1.299×
4	**Reference Baseline**	**1.370**×
6	Redundancy	1.261×
8	Redundancy	1.231×
Static	–	1.219×

5.3 Further Analysis

5.3.1 Comparison of Profiling Feedback Strategies

We evaluate five distinct control conditions over a representative subset of 48 diverse kernels spanning across all difficulty levels: (1) *Blind*: provides no profiling metadata; (2) *Random*: attaches a randomly sampled profiling configuration; (3) *Static*: uses a fixed, globally optimal profiling setup; (4) *AutoProfiler*: dynamically assigns kernel-adaptive profiling trees; and (5) *Oracle*: assigns the empirical upper-bound configuration for each individual kernel, which is unavailable during inference. All other components, including the rewriter, optimization pipeline, hardware, and sampling seeds, are kept identical.

As summarized in Table 2, profiling feedback consistently improves kernel optimization. Our *AutoProfiler* achieves a 1.379× speedup, compared with 1.197× for *Blind*, and also outperforms both *Random* and *Static*. These results show that the benefit comes not only from providing profiling information, but from selecting feedback adaptively for each kernel. The remaining gap to *Oracle* suggests further room for improving profiling selection.

5.3.2 More Profiling Is Not Always Better

We further investigate how the volume of profiling feedback influences downstream optimization efficacy. Starting from the learned 4-metric setup as our reference, we construct reduced variants by pruning metrics and expanded ones by sampling additional metrics from the NCU pool. As presented in Table 3, downstream performance exhibits a clear non-monotonic trend with respect to profiling metric quantity. Reducing feedback from four metrics to two drops the geomean speedup from 1.370× to 1.299×. Conversely, over-saturating the context with six or eight metrics degrades performance further to 1.261× and 1.231×, respectively. These results demonstrate that effective profiling requires judicious metric selection to maximize information density, avoiding both metric scarcity and redundancy.

5.3.3 Generalization Across LLM Backbones

To test whether the learned profiling policy captures intrinsic hardware requirements rather than overfitting to a specific LLM, we evaluate its transferability across model families. Specifically, we freeze the profiling tree trained on GPT-5.6-Sol and replace only the leaf kernel-rewriting LLM backbone with five distinct models, evaluating each on held-out benchmarks under the same pipeline.

As summarized in Table 4, the frozen profiling policy generalizes strongly across all backbones. When paired with Claude-Sonnet-5, the system achieves a peak geomean speedup of 1.47× and an execution accuracy of 91.7%, outperforming the training time baseline. Furthermore, across backbones from diverse model families (*e.g.,* Kimi-K3, DeepSeek-V4-Pro, and Qwen3-Coder-Plus), the profiling policy consistently guides effective kernel optimization. These results confirm that the learned profiling tree encodes hardware optimization priors that are independent of the underlying LLM backbone rather than model specific biases.

5.3.4 Iterative Kernel Refinement

We evaluate multi-turn optimization by recursively applying the frozen profiling tree over ten consecutive rounds, using the kernel produced in each step as the input for the next. Figure 3 reports the

Table 4: Generalization performance across different LLM backbones. The profiling tree is constructed using GPT-5.6-Sol feedback and kept strictly frozen; only the leaf LLM is swapped at test time. Speedup denotes the geometric mean over PyTorch baselines on held-out kernels.

Rewriter	Model Family	Speedup (↑)	Accuracy (↑)	Time (s, ↓)	Tokens (↓)
Claude-Sonnet-5	Anthropic	1.47×	91.7	8.2	1,840
Kimi-K3	Moonshot	1.38×	89.6	9.1	2,010
GPT-5.6-Sol	OpenAI	1.34×	87.5	10.4	2,260
DeepSeek-V4-Pro	DeepSeek	1.21×	83.3	12.7	2,580
Qwen3-Coder-Plus	Qwen	1.12×	79.2	14.3	3,120

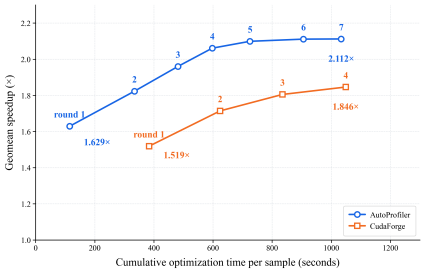


Figure 3: Comparison of optimization trajectories. Each circled marker represents an optimization round.

Configuration	Depth	# Leaves	Speedup (↑)
AutoProfiler	11	58	**1.61**×
w/o REWRITE	2	4	1.47×
w/o SPLIT	7	37	1.53×

Table 5: Ablation study on tree-construction operators.

optimization trajectory as a function of cumulative wall-clock time per sample, providing a direct comparison with CudaForge under actual optimization cost. AutoProfiler improves rapidly from 1.629× in the first round to 2.112× after seven rounds, while requiring substantially less time per iteration. In comparison, CudaForge reaches 1.846× after four rounds under a similar cumulative time budget. These results show that the learned profiling policy not only supports stable iterative refinement, but also enables a more favorable optimization-cost trade-off by avoiding repeated expensive profiling and LLM-based profiling decisions.

5.4 Ablation Study

As summarized in Table 5, removing either operator causes a clear performance drop, demonstrating that both components are essential for optimal tree construction. Removing REWRITE drops the geomean speedup from 1.61× to 1.47×. In this configuration, while SPLIT still divides kernels into separate branches, each leaf provides only basic metric feedback. Without the enriched profiling metrics, the downstream LLM lacks sufficient hardware context to guide fine-grained optimizations. Conversely, removing SPLIT lowers performance to 1.53×. Without adaptive node splitting, the tree cannot separate kernels with distinct bottlenecks, forcing very different workloads to share the same profiling setup. The full policy achieves the peak speedup of 1.61× by combining both operators. These results confirm that SPLIT groups kernels into specialized hardware regimes, while REWRITE ensures each regime receives the precise profiling details needed for LLM-driven code optimization.

6 Conclusion

In this paper, we present AutoProfiler, a novel feedback-driven framework that addresses the fundamental limitations of fixed hardware profiling in LLM-based GPU kernel optimization. Motivated by the observation that less profiling can yield better optimization, we shift from static metric collection to a profiling reasoning perspective, AutoProfiler learns a deterministic, kernel-conditioned profiling policy via iterative tree construction using REWRITE and SPLIT operations. Empirical results on KernelBench show that AutoProfiler outperforms traditional static profiling configurations and strong agentic baselines under constrained profiling budgets while incurring minimal LLM invocation overhead. Our findings suggest that hardware profiling should be treated as a learnable decision process, opening a promising direction for future research in AI-driven systems optimization.

AI USE STATEMENT

In this work, we used generative AI tools for writing refinement, text polishing, and grammatical editing during manuscript preparation. We have not used generative AI tools for research ideation, theoretical derivations, baseline implementations, or experimental evaluations, and the rest of the required disclosure tasks are not applicable to this work. Additionally, we used generative AI tools for formatting suggestions and phrasing improvements. We have reviewed all AI-assisted work: all revised text, technical descriptions, and mathematical notations were manually verified by the authors to ensure factual accuracy and consistency with our methodology. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We provide detailed descriptions of the profiling configurations, evaluation metrics, and hardware setup in Section 5.1 and additional empirical analyses in the Appendix A.2. All experiments use fixed hardware environments and controlled random seeds where applicable. We will release the code after the review period.

REFERENCES

Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. Kevin: Multi-turn rl for generating cuda kernels. In *International Conference on Learning Representations (ICLR)*, 2026.

Martin Burtscher et al. Perfexpert: An easy-to-use performance diagnosis tool for hpc applications. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.

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.

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 (NeurIPS)*, volume 31, 2018b.

Kaiming Cheng, Laura Wang, Jack Khuu, Mark Saroufim, Wenyuan Chi, Jiannan Wang, and Joe Isaacson. Kernelagent: Hardware-guided gpu kernel optimization via multi-agent orchestration. *arXiv preprint*, 2026.

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.

Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Masahiro Masuda, Cody Hao Yu, and Tianqi Chen. Tensorir: An abstraction for tensor program program transformation. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, volume 2, pp. 801–816, 2023.

Philip J Fleming and John J Wallace. How not to lie with statistics: the correct way to summarize benchmark results. *Communications of the ACM*, 29(3):218–221, 1986.

Jiading Gai, Shuai Zhang, Kaj Bostrom, Jin Huang, Vihang Patil, Haoyang Fang, Yuyang Wang, Huzefa Rangwala, and George Karypis. Optimizing cuda like a human: Micro-profiling tools as expert surrogates for llm-based gpu kernel optimization. *arXiv preprint arXiv:2606.26453*, 2026a.

Jiading Gai, Shuai Zhang, Kaj Bostrom, Jin Huang, Vihang Patil, Haoyang Fang, Yuyang Wang, Huzefa Rangwala, and George Karypis. Optimizing cuda like a human: Micro-profiling tools

as expert surrogates for llm-based gpu kernel optimization. *arXiv preprint arXiv:2606.26453*, 2026b.

Jaber Jaber and Osama Jaber. Autokernel: Autonomous gpu kernel optimization via iterative agent-driven search. *arXiv preprint arXiv:2603.21331*, 2026.

NVIDIA Corporation. Nvidia nsight compute profiling guide. `https://docs.nvidia.com/nsight-compute/ProfilingGuide/`, 2023.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Re, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, volume 267, pp. 47356–47415, 2025.

Dezhi Ran, Shuxiao Xie, Mingfang Ji, Ziyue Hua, Mengzhou Wu, Yuan Cao, Yuzhe Guo, Hao Yu, Linyi Li, Yitao Hu, and Tao Xie. Kernelband: Boosting llm-based kernel optimization with a hierarchical and hardware-aware multi-armed bandit. *CoRR*, abs/2511.18868, 2025.

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 (NeurIPS)*, volume 35, pp. 35783–35796, 2022.

Qitong Sun, Jun Han, Tianlin Li, Zhe Tang, Sheng Chen, Fei Yang, Aishan Liu, Xianglong Liu, and Yang Liu. Kernelskill: A multi-agent framework for gpu kernel optimization. *arXiv preprint arXiv:2603.10085*, 2026.

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

Yunxiang Zhang, Ping Yu, Jianyu Wang, Max Fan, Julian Reed, Azalia Mirhoseini, and Will Su. Kernelbench-verified: Do llm-generated kernels actually beat pytorch? *CoRR*, abs/2607.16241, 2026.

Zijian Zhang, Rong Wang, Shiyang Li, Yuebo Luo, Mingyi Hong, and Caiwen Ding. Cudaforge: An agent framework with hardware feedback for cuda kernel optimization. *arXiv preprint arXiv:2511.01884*, 2025.

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

Size Zheng, Yun Liang Lin, Cheng Si Zhang, Xiuhong Ren, Yun-Sheng Chen, Lingxiao Huang, Yun Liang, and Xuechao Ning. Flextensor: An automatic schedule exploration framework for tensor computation on heterogeneous hardware. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 859–873, 2020b.

A Appendix

A.1 Related Work

Traditional GPU Kernel Optimization. Automated GPU kernel optimization has long relied on search-based compilation frameworks and hardware-aware cost models. Early systems like TVM (Chen et al., 2018a) and AutoTVM (Chen et al., 2018b) introduced learned cost models to guide program space exploration. Ansor (Zheng et al., 2020a) expanded this search space via hierarchical program sampling, while MetaSchedule (Shao et al., 2022) unified rule-based and search-based program transformations. Further extensions such as FlexTensor (Zheng et al., 2020b) and TensorIR (Feng et al., 2023) specialized in tensor program abstractions. While effective, these compiler-based

approaches operate on rigid, template-driven program spaces and treat hardware profiling primarily as passive static evaluation or fixed performance metrics, lacking the adaptive code reasoning capabilities offered by modern large language models.

LLM-driven Kernel Optimization and Agentic Frameworks. With the rise of LLMs, recent efforts leverage generative models for GPU kernel synthesis and iterative optimization. Benchmarks like KernelBench (Ouyang et al., 2025) have formalized the evaluation of LLM-generated CUDA and Triton kernels. Building on this, agentic systems integrate execution and runtime feedback into iterative search loops. For instance, CudaForge (Zhang et al., 2025) and AutoKernel (Jaber & Jaber, 2026) apply iterative compilation and error-repair heuristics, whereas KernelAgent (Cheng et al., 2026) and Astra (Wei et al., 2025) employ multi-agent orchestration for kernel refactoring. Additionally, Kevin (Baronio et al., 2026) and CUDA Agent (Dai et al., 2026) explore reinforcement learning and dynamic tool invocation to guide kernel synthesis. Despite their success, these frameworks treat hardware profiling as an invariant preprocessing step or a static feedback prompt. They overlook the fact that hardware profiling is inherently costly and that metric utility varies across heterogeneous GPU kernels.

Hardware Profiling: From Static Metric Collection to Profiling Reasoning. Hardware profiling tools such as NVIDIA Nsight Compute (NVIDIA Corporation, 2023) provide hundreds of microarchitectural performance counters, but full metric acquisition incurs non-trivial latency and context redundancy. Early performance diagnosis tools like PerfExpert (Burtscher et al., 2010) used static heuristic trees to isolate performance bottlenecks. Recent LLM-based micro-profiling assistants, such as KernelPro (Gai et al., 2026b), selectively trigger profiling tools based on predefined rule sets. However, existing methods either collect a universal metric subset indiscriminately or rely on hand-crafted profiling rules. In contrast, our work introduces the paradigm of *profiling reasoning*: we formulate hardware metric acquisition as a kernel-conditioned sequential decision process, explicitly learning a deterministic profiling policy driven directly by downstream code optimization rewards.

A.2 Empirical Verification of Profiling-Utility Heterogeneity

To empirically validate the heterogeneity assumption underlying Proposition 1, we quantify the performance discrepancy between static metric selection and dynamic, per-kernel metric selection on a representative set of 128 kernels sampled from KernelBench.

Candidate Configurations and Profiling Setup. We construct four static candidate profiling configurations, denoted as $\mathcal{C} = \{m_{\text{algo}}, m_{\text{mem}}, m_{\text{lat}}, m_{\text{comp}}\}$, designed to capture distinct, roofline-relevant execution characteristics:

- **Algorithm** (m_{algo}): Captures high-level algorithmic behavior and control-flow patterns.
- **Memory** (m_{mem}): Focuses on DRAM/SRAM memory bandwidth utilization and transaction efficiency.
- **Latency** (m_{lat}): Measures instruction pipeline stalls, execution latency, and warp scheduling overheads.
- **Compute** (m_{comp}): Tracks compute unit saturation, floating-point throughput, and ALU utilization.

All metric sets are fixed prior to evaluation. For each kernel $k \in \mathcal{K}$ and candidate configuration $m_i \in \mathcal{C}$, we extract the profiling information $I_i(k) = P(k, m_i)$ and invoke a frozen kernel rewriter $g(\cdot)$ under identical optimization, hardware, and random seed settings. We record the resulting speedup relative to the PyTorch reference implementation as $s_i(k) = s(k, m_i)$.

Static Benchmark vs. Per-Kernel Oracle. We compare the performance of the single best static configuration against an empirical per-kernel oracle. The optimal fixed configuration m^* is chosen to maximize the geometric mean speedup across the benchmark:

$$m^* = \underset{m_i \in \mathcal{C}}{\operatorname{argmax}} \exp\left(\frac{1}{|\mathcal{K}|} \sum_{k \in \mathcal{K}} \ln s_i(k)\right). \tag{4}$$

Conversely, the per-kernel oracle dynamically selects the optimal configuration independently for each individual kernel:

$$s_{\text{oracle}}(k) = \max_{m_i \in \mathcal{C}} s_i(k). \tag{5}$$

Kernels for which no metric configuration yields a speedup over the unoptimized reference implementation ($s_i(k) \leq 1.0$) are tracked separately under non-improving cases.

Empirical Findings and Headroom Analysis. As detailed in Table 6, no static configuration consistently dominates across heterogeneous workloads. While the strongest fixed baseline Algorithm achieves a geometric mean speedup of $1.332\times$, the per-kernel oracle reaches $1.429\times$—delivering a 7.3% relative performance improvement.

To assess statistical confidence, we conduct $1,000$ bootstrap resamples over the 128 evaluation kernels, establishing a 95% confidence interval of $[1.045\times, 1.107\times]$ for the relative gain of the oracle over the static baseline. Furthermore, the optimal profiling configuration shifts substantially across workloads: Memory provides the highest speedup for 31 kernels, Latency for 27, Algorithm for 26, and Compute for 23. These empirical observations highlight two key insights:

1. **High Utility Heterogeneity:** The value of profiling metrics is fundamentally kernel-dependent, rendering any fixed, one-size-fits-all profiling configuration inherently suboptimal.
2. **Substantial Optimization Headroom:** Tailoring hardware profiling signals to specific kernel bottlenecks yields meaningful execution gains, underscoring the need for an adaptive profiling policy like AUTOPROFILER.

Table 6: Empirical heterogeneity of profiling utility on 128 KernelBench kernels. Each candidate configuration is evaluated with the same frozen kernel optimizer.The "Best on # Kernels" column counts kernels for which the configuration achieves the highest downstream speedup among the four candidates. The oracle selects the best candidate independently for each kernel.

Profiling Configuration	Geomean Speedup	Best on # Kernels
Algorithm	$1.332\times$	26
Memory	$1.277\times$	31
Latency	$1.242\times$	27
Compute	$1.239\times$	23
Per-kernel Oracle	**1.429**$\times$	128