EVOCOMPILE: BRIDGING THE COMPILATION EFFECTIVENESS GAP THROUGH EXPERIENCE-ADAPTIVE AGENTIC OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Modern deep learning compilers provide powerful optimization capabilities, yet effectively utilizing them across diverse training workloads still requires workload-specific decisions. Different workloads may expose insufficient computation to the compiler, contain compiler-unfriendly program structures, or leave additional optimization opportunities unexplored. We characterize this problem as the *compilation-effectiveness gap*, defined as the performance gap between general compiler usage and workload-specific optimization. We present **EvoCompile**, an experience-adaptive agentic optimization framework that reduces this gap by learning how workloads should be presented to existing compilers. EvoCompile accumulates reusable optimization experience and applies targeted interventions across compilation, program, and kernel levels, refined through execution feedback. We evaluate EvoCompile on 119 workloads from HuggingFace, TIMM, and TorchBench, together with large-scale training and cross-backend scenarios. EvoCompile consistently improves compilation effectiveness while preserving correctness, achieving a **1.89**× geometric-mean speedup on TorchBench. Code is available at `https://anonymous.4open.science/r/EvoCompile-8C20/`

1 INTRODUCTION

Modern deep learning training increasingly relies on compiler technologies to improve execution efficiency. Systems such as PyTorch `torch.compile` (Ansel et al., 2024; Wen, 2023) combine dynamic graph capture and hardware-aware code generation, substantially reducing the effort required to optimize high-level training programs. However, enabling compilation does not always translate into effective acceleration. The achievable performance depends not only on the compiler itself, but also on how a workload is exposed to it: compilation boundaries, Python execution logic, optimizer operations, and dynamic behaviors can determine which computations are captured and optimized. Graph breaks, for example, can fragment computation and reduce available optimization opportunities (Ansel et al., 2024; Kashmira et al., 2025).

We refer to the performance gap between general compiler usage and workload-specific optimization as the *compilation-effectiveness gap*. In practice, this gap can arise when dominant computation falls outside the compiled region, program structures inhibit effective graph capture, or additional optimization opportunities remain beyond the default compilation workflow. Existing approaches address only parts of this problem. Autotuning systems search predefined optimization spaces (Chen et al., 2018b; Zheng et al., 2020), while workload-adaptation techniques typically target specific mechanisms such as graph-break repair (Kashmira et al., 2025). Consequently, they do not systematically determine and refine how different workloads should be presented to existing compilers. This motivates a broader question: *Can compiler optimization experience be accumulated and reused to automatically adapt compilation strategies to new workloads?*

We present **EvoCompile**, an experience-adaptive agentic optimization framework for reducing the compilation-effectiveness gap. Rather than modifying compiler internals, EvoCompile observes workload and compilation behavior, reuses validated optimization experience, and applies workload-specific interventions across compilation, program, and kernel levels. Execution feedback validates

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.

each intervention and updates the accumulated experience, enabling subsequent workloads to benefit from previous optimization outcomes.

We evaluate EvoCompile on 119 training workloads from HuggingFace, TIMM, and TorchBench, together with large-scale training and cross-backend scenarios. With GPT-5.6-Sol, EvoCompile achieves geometric-mean speedups of **2.14**×, **2.01**×, and **1.89**× on HuggingFace, TIMM, and TorchBench, respectively, compared with 2.03×, 1.99×, and 1.77× under the default benchmark compilation workflow. It also achieves optimization success rates of **100%**, **100%**, and **94%** across the three suites. These results show that systematically adapting workload presentation can further improve the effectiveness of existing compiler capabilities.

Our contributions are summarized as follows:

- We formulate the compilation-effectiveness gap between general compiler usage and workload-specific optimization, and characterize its sources in practical deep learning training workloads.
- We present EvoCompile, an experience-driven optimization framework that reduces this gap through workload-aware experience reuse and structured multi-level interventions without modifying compiler internals.
- We conduct extensive evaluations across benchmark workloads, large-scale training scenarios, and different compilation backends, showing that EvoCompile consistently improves compilation effectiveness while preserving correctness.

2 Related Work

Existing deep learning compilers optimize execution through graph transformation, scheduling, and code generation (Google Research, 2020), while modern frameworks such as PyTorch 2 (Ansel et al., 2024) further support dynamic program compilation through TorchDynamo, AOTAutograd, and TorchInductor. Recent work has explored workload adaptation, including eliminating graph breaks through program rewriting (Kashmira et al., 2025) and applying regional compilation in large-scale training (Liang et al., 2024), as well as LLM-assisted compiler optimization (Cummins et al., 2023). However, existing approaches are largely limited to predefined adaptation mechanisms or per-workload optimization, and do not systematically accumulate reusable knowledge across workloads. In contrast, EvoCompile continuously converts workload characteristics, optimization interventions, and execution outcomes into reusable experience for guiding future optimization. We provide an extended discussion of related work in Appendix A.

3 EvoCompile

3.1 Problem Definition

We study the problem of reducing the performance gap between general compiler usage and workload-specific optimization for practical deep learning training workloads. Unlike compiler design that modifies the optimization engine itself, EvoCompile focuses on discovering how a workload should be presented to an existing compiler to better utilize its available optimization capability. We define a training workload as $\mathcal{W} = (M, D, O, C)$, where M denotes the model implementation, D denotes the training data and dataloader, O denotes the optimization procedure, and C represents runtime and compilation configurations. Given a workload $\mathcal{W}$, a general compilation strategy ϕ_0 produces an executable program:

$$P_0 = \phi_0(\mathcal{W}). \tag{1}$$

Although modern compilers can aggressively optimize captured regions, ϕ_0 is designed for broad compatibility and may not expose the most effective optimization opportunities for individual workloads. We define the workload-specific optimization space Φ as possible compilation strategies, including region selection, program adaptation, configuration adjustment, and additional transformations. The performance difference between general compilation and an effective workload-specific strategy $\phi^* \in \Phi$ is defined as the *compilation-effectiveness gap*:

$$G(\mathcal{W}) = \frac{T(\phi_0(\mathcal{W})) - T(\phi^*(\mathcal{W}))}{T(\phi_0(\mathcal{W}))}. \tag{2}$$

As defined in Eq. 2, a larger $G(\mathcal{W})$ indicates that existing compiler capabilities are not fully utilized because general compilation strategies cannot adapt to workload-specific characteristics. The goal of EvoCompile is therefore to automatically identify workload-specific compilation strategies that reduce this gap while preserving training correctness.

3.2 Overview

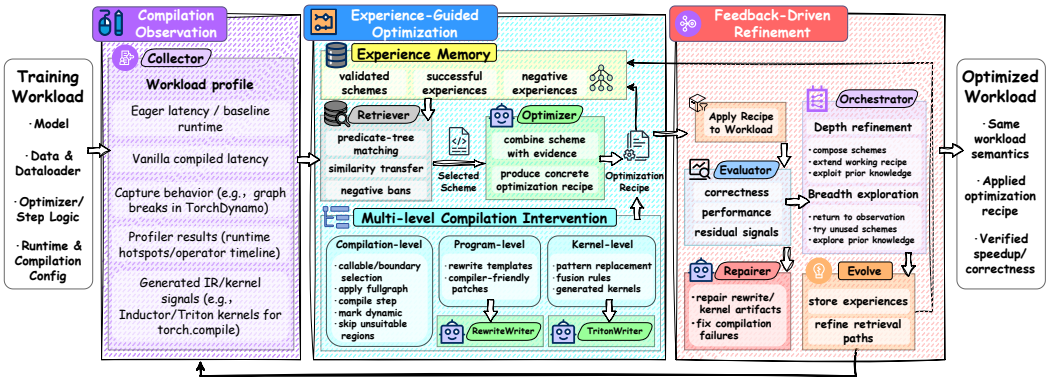

Figure 1: Overview of EvoCompile. Given a training workload, EvoCompile collects compilation evidence, retrieves validated optimization schemes, and generates workload-specific optimization recipes. The recipes perform multi-level interventions across compilation, program, and kernel levels, while execution feedback continuously refines optimization trajectories and experience memory.

To bridge the compilation-effectiveness gap, EvoCompile treats compilation optimization as a closed-loop experience refinement process. Given a training workload, EvoCompile observes its compilation behavior, retrieves relevant optimization knowledge, applies workload-specific interventions, and refines future decisions based on execution feedback. The optimization loop consists of three stages: ❶ **Compilation Observation.** EvoCompile collects compiler and runtime evidence, including graph capture behavior, execution profiles, and workload characteristics, to characterize how the workload is currently exposed to the compiler. ❷ **Experience-guided optimization.** EvoCompile retrieves accumulated optimization experience and instantiates workload-specific optimization recipes based on validated schemes and current workload evidence, rather than relying on fixed compilation policies. ❸ **Feedback-driven refinement.** EvoCompile evaluates applied recipes under correctness and performance constraints and updates its experience memory using both successful and failed outcomes. This closed-loop design enables EvoCompile to progressively accumulate reusable compiler optimization knowledge across workloads. The overall framework of EvoCompile is illustrated in Figure 1.

3.3 Experience-Adaptive Agentic Optimization

We next detail how EvoCompile implements this closed-loop optimization process. Specifically, we introduce the components responsible for compilation observation, experience-guided optimization, and feedback-driven refinement, which collaboratively convert workload evidence into optimization recipes and continuously improve them through execution feedback.

3.3.1 Compilation Observation

EvoCompile first collects workload-specific evidence that characterizes the current compilation behavior. Instead of directly modifying the training program, the Collector module constructs an execution profile through lightweight observation. The collected evidence includes eager and vanilla compilation latency, graph-break statistics, representative break locations, profiler results, and generated kernel information such as Inductor external calls and Triton kernel names.

These observations characterize how the workload is exposed to the compiler and provide evidence for subsequent optimization decisions. The collected workload profile is passed to the retrieval and planning stages. Importantly, this stage only records compilation behavior without introducing optimization assumptions or modifying the original workload.

3.3.2 EXPERIENCE-GUIDED OPTIMIZATION

After obtaining the workload profile, EvoCompile determines how existing compiler capabilities can be better utilized for the current workload. However, raw observations, such as graph-break patterns or kernel profiles, do not directly specify optimization actions. Therefore, EvoCompile leverages accumulated compiler experience to transform workload evidence into executable optimization decisions.

The Retriever module converts observations into structured retrieval keys, including execution latency, graph-break characteristics, representative break locations, profiler statistics, and kernel information. These keys are used to traverse a hierarchical predicate-based retrieval tree to identify candidate optimization schemes matching the current workload characteristics. Similarity-based retrieval further augments hard matching by transferring nearby successful experiences and using negative experiences to avoid ineffective strategies. Each scheme represents a validated optimization pattern accumulated from previous workloads and specifies the applicable intervention scope. Given a selected scheme, the Optimizer module instantiates it into a workload-specific optimization recipe. The Optimizer does not freely decide which aspect of the workload should be modified; instead, it fills concrete operations according to the selected scheme and current workload evidence.

This design avoids unconstrained modifications by a general-purpose LLM. Although an LLM can generate arbitrary code changes, such modifications may not correspond to the actual compilation bottleneck or may violate workload correctness. By grounding optimization decisions on validated schemes and workload-specific evidence, EvoCompile generates targeted and reliable optimization recipes. The generated recipe defines executable optimization operations, which are implemented through three complementary intervention levels described in Section 3.4.

3.3.3 FEEDBACK-DRIVEN REFINEMENT

Retrieved experiences provide an effective initialization for optimization, but they cannot guarantee that a selected scheme completely resolves the compilation-effectiveness gap of a new workload. The experience memory is constructed from previously observed cases, and retrieval relies on workload similarity. Therefore, a retrieved scheme may require further adaptation, and additional optimization opportunities may remain after applying the initial recipe.

To address this limitation, EvoCompile uses execution feedback to refine both the current optimization trajectory and the accumulated experience memory. Refinement does not treat the optimized workload as a new case. Instead, EvoCompile preserves the original workload identity and evaluates every recipe against the frozen vanilla baseline. During execution, the system records residual compilation signals produced by the applied recipe, which reveal whether complementary optimization opportunities remain.

The refinement process consists of four components: Evaluator, Repairer, Orchestrator, and Evolve. The Evaluator verifies correctness, measures performance improvement, and provides residual signals such as remaining graph-break patterns or uncovered kernel opportunities. Based on these signals, the Orchestrator balances exploitation of existing optimization knowledge and exploration of alternative strategies. For exploitation, EvoCompile follows a depth-oriented refinement process. When a recipe executes successfully, the Orchestrator checks whether residual signals match complementary schemes that have not been applied. If such schemes exist, their operations are composed with the current recipe and evaluated again on the original workload. This enables EvoCompile to progressively extend a working optimization trajectory by combining multiple optimization opportunities. For exploration, when the current trajectory cannot be further extended, EvoCompile returns to the original workload observation and selects alternative unused schemes. This breadth-oriented exploration is triggered when a recipe fails, cannot be repaired, or cannot be extended with compatible schemes. Unlike unrestricted program search, exploration remains constrained within validated optimization schemes maintained by the experience memory. During optimization exploration, individual interventions may fail due to workload-specific constraints. The Repairer handles such fail-

ures without replacing the selected optimization strategy. For compilation-related failures, Repairer adjusts compilation-level decisions, such as relaxing `fullgraph` constraints or disabling incompatible regions. For generated artifacts, RewriteWriter and TritonWriter perform rewrite-specific and kernel-specific repairs, respectively.

Finally, the Evolve module updates the experience memory using both successful and unsuccessful outcomes. Successful optimization trajectories are stored as reusable experiences, while failed attempts are preserved as negative experiences that constrain future retrieval and exploration. Through this feedback-driven refinement process, EvoCompile transforms static experience retrieval into an adaptive optimization loop. Retrieved schemes provide initial guidance, execution feedback reveals remaining opportunities, and accumulated outcomes continuously improve future optimization decisions.

3.4 Multi-level Compilation Intervention

The generated optimization recipe defines executable operations that determine how workloads are exposed to existing compilation systems. Since compilation-effectiveness gaps arise from different sources, EvoCompile performs interventions at three complementary levels: compilation, program, and kernel. These levels define the action space of optimization recipes rather than independent optimization stages. Importantly, EvoCompile does not modify compiler internals; instead, it improves compiler utilization by adapting workload presentation or extending available optimized operators.

❶ **The compilation level**, EvoCompile adjusts which existing computation regions are exposed to the compiler and defines appropriate compilation boundaries. Typical operations include selecting target callables, changing compilation scopes, applying `fullgraph` to suitable regions, compiling optimizer or training-step components, enabling dynamic shape annotations, or skipping unsuitable regions. These operations address cases where existing compiler capabilities are available but the default compilation scope fails to expose beneficial computation.

❷ **The program level**, EvoCompile applies semantics-preserving adaptations when workload structures prevent effective graph capture. Most operations are instantiated from existing rewrite templates, such as class-level forward replacement or framework-specific adjustments. For previously unseen patterns, RewriteWriter generates required in-memory patches. These adaptations expose compiler-friendly execution structures without changing the training objective.

❸ **The kernel level**, EvoCompile addresses operator patterns that remain inefficient under the current compilation workflow. Many kernel-level operations reuse predefined replacement or fusion patterns. When a required pattern is unavailable, TritonWriter generates a new kernel implementation and registers it before compilation, allowing the compiler pipeline to incorporate the additional optimization. Unlike modifying generated compiler outputs, kernel-level interventions extend the available optimized operators before compilation. Further implementation details are provided in Appendix B.

A single recipe may contain operations from multiple intervention levels, such as combining a compilation-boundary adjustment with a program rewrite. RewriteWriter and TritonWriter only materialize operations that require generated artifacts; they do not determine the optimization strategy.

4 Experiments

4.1 Benchmark and Metrics

We evaluate EvoCompile on 119 workloads from the PyTorch benchmark ecosystem (Paszke et al., 2019), including 30 HuggingFace, 18 TIMM, and 71 TorchBench cases after official metadata-based filtering. The original benchmark configurations are kept unchanged. We additionally evaluate EvoCompile on large-scale training scenarios and different compilation backends.

We use two metrics.

Speedup measures acceleration relative to eager execution:

$$Speedup = \frac{T_{eager}}{T_{method}},$$

where T_{eager} and T_{method} denote the training step latency of eager execution and the evaluated method, respectively. We report the geometric mean speedup over each benchmark suite.

Success measures the ratio of workloads that achieve speedup over eager execution while maintaining numerical consistency:

$$Success = \frac{N_{success}}{N_{total}}.$$

4.2 Baselines

We compare EvoCompile with existing compilation strategies and a general-purpose coding agent.

Naive compile. We introduce a naive compilation baseline by directly applying `torch.compile(model)` to the default training entry point, following the standard PyTorch compilation interface (Ansel et al., 2024). This baseline represents the most straightforward usage of compiler acceleration without workload-specific optimization.

Benchmark Compilation (Default). We compare against the default compilation workflow adopted by the PyTorch benchmark suite (Paszke et al., 2019). This workflow follows the standard configuration with TorchDynamo-based graph capture and TorchInductor backend compilation.

Compiler Optimization Modes. Based on the benchmark compilation workflow, we further evaluate built-in compiler modes, including `reduce-overhead` and `max-autotune` provided by PyTorch compilation (Wen, 2023). These modes adjust compiler optimization behaviors within the standard compilation pipeline.

General-purpose Coding Agent. We compare against Cursor (Cursor Team, 2025), a widely used general-purpose LLM-based coding agent. This baseline evaluates whether a framework specifically designed to reduce the compilation-effectiveness gap provides additional benefits over general-purpose agents.

4.3 Setting

All experiments are conducted on NVIDIA A100 GPUs with Python 3.8, PyTorch 2.4.1, and CUDA 12.1. We evaluate EvoCompile on HuggingFace, TIMM, and TorchBench workloads following the official PyTorch benchmark configurations. The workload input settings, including model configurations and input sizes, are kept unchanged from the original benchmark suite. For each workload, EvoCompile performs 10 optimization iterations to progressively refine compilation strategies. After optimization, we measure execution latency using 10 warmup iterations followed by 40 measured iterations.

The correctness of optimized workloads is verified by comparing their outputs with eager execution using numerical agreement checking with `torch.allclose`, where both absolute and relative tolerances are set to 10^{-2}.

To evaluate robustness, we instantiate EvoCompile with different LLM backbones, including Qwen3.8-Coder and ChatGPT-5.6-Sol, while keeping the optimization framework unchanged. We additionally evaluate EvoCompile across different compilation backends and large-scale training scenarios.

4.4 Results and Analysis

4.4.1 Overall Performance

Table 1 summarizes the comparison between EvoCompile and existing compilation strategies and general-purpose agent baselines on PyTorch benchmark workloads.

EvoCompile consistently achieves higher geometric mean speedup across HuggingFace, TIMM, and TorchBench. Compared with naive torch.compile, EvoCompile improves the speedup from

Table 1: Overall performance comparison on PyTorch benchmark workloads. Each entry reports the geometric mean speedup over all workloads in the corresponding benchmark suite relative to eager execution, together with the Success.

Method	HuggingFace		TIMM		TorchBench	
	Speedup	Success	Speedup	Success	Speedup	Success
Naive Compile	1.68	1.00	1.73	0.89	1.24	0.68
Benchmark Compile(Default)	2.03	1.00	1.99	1.00	1.77	0.87
Reduce-Overhead	2.04	1.00	1.99	1.00	1.77	0.89
Max-Autotune	2.04	1.00	2.00	1.00	1.78	0.89
Cursor(GPT-5.6-Sol)	1.76	1.00	1.83	1.00	1.26	0.70
EvoCompile(Qwen3.8-Coder)	**2.07**	**1.00**	**2.00**	**1.00**	**1.87**	**0.92**
EvoCompile(GPT-5.6-Sol)	**2.14**	**1.00**	**2.01**	**1.00**	**1.89**	**0.94**

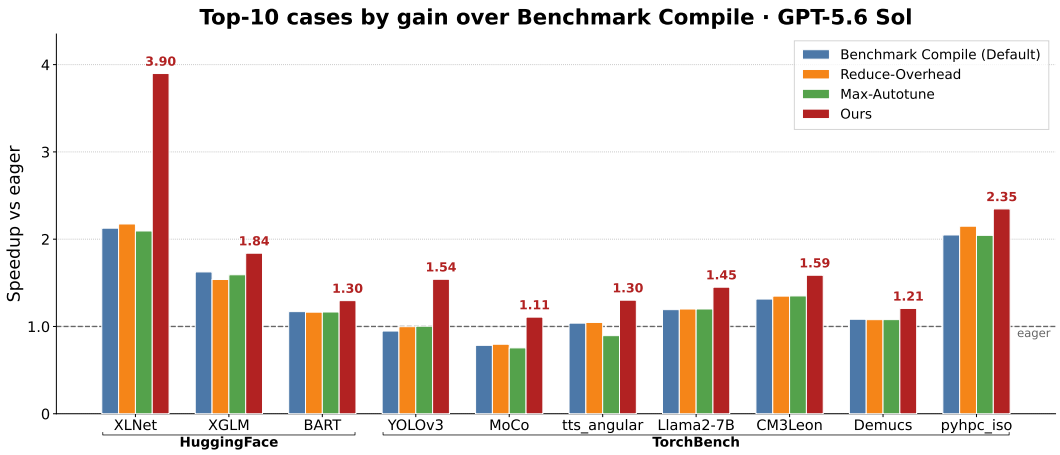

Figure 2: Top-10 workloads with the largest improvements over the default PyTorch benchmark compilation workflow. The results illustrate workload-specific optimization opportunities discovered by EvoCompile.

1.68× to 2.14× on HuggingFace, from 1.73× to 2.01× on TIMM, and from 1.24× to 1.89× on TorchBench. These results demonstrate that workload-specific optimization decisions can further improve compiler effectiveness beyond direct compilation.

Compared with the official benchmark compilation workflow and built-in compiler modes, including `reduce-overhead` and `max-autotune`, EvoCompile achieves consistent improvements, particularly on TorchBench workloads with diverse architectures and execution behaviors. Compared with Cursor, a general-purpose coding agent, EvoCompile achieves higher optimization effectiveness. The results suggest that addressing the compilation-effectiveness gap requires optimization strategies specifically designed around compiler workload presentation, including reusable compilation adaptation experience and structured intervention mechanisms.

We evaluate EvoCompile with two LLM backbones, GPT-5.6-Sol and Qwen3.8-Coder. Both variants achieve consistent improvements, indicating that the effectiveness of EvoCompile is not dependent on a specific language model backbone.

Figure 2 further shows the workloads with the largest gains over the official benchmark compilation strategy, illustrating the workload-dependent optimization opportunities addressed by EvoCompile.

Table 2: Large-scale training comparison between TorchTitan-based configurations and EvoCompile. Latency denotes the training step time in milliseconds (ms), and speedup is measured relative to eager execution.

Model	Config.	Eager (ms)	TorchTitan		EvoCompile	
			Latency (ms)	Speedup	Latency (ms)	Speedup
Qwen3 MoE	S=1k, B=2	249	200	1.25×	**178**	**1.40**×
DeepSeek-V3 MoE	S=1k, B=2	115	85	1.35×	**83**	**1.39**×
Llama3-8B	S=1k, B=2	2256	2171	1.04×	**2170**	**1.04**×
Qwen3 MoE	S=2k, B=2	325	249	1.31×	**236**	**1.38**×
DeepSeek-V3 MoE	S=2k, B=2	159	135	1.18×	135	1.18×
Llama3-8B	S=2k, B=2	3622	3486	1.04×	3487	1.04×

Table 3: Cross-backend generalization of EvoCompile. Each entry reports the geometric mean speedup of EvoCompile over the corresponding default compilation entry of each backend.

Backend	Compilation Entry	HuggingFace	TIMM	TorchBench
Inductor	`torch.compile(model)`	1.27	1.09	1.47
OpenXLA	`torch_xla.compile(step)`	1.36	1.39	1.23
Thunder	`thunder.jit(model)`	1.32	4.34	2.23

4.4.2 Large-scale Training Case Study

We evaluate EvoCompile on three large-scale language model training scenarios and compare it with TorchTitan-based training configurations. The evaluated models include Qwen3-MoE and DeepSeek-V3-MoE debug variants with approximately 0.32B and 33M parameters, respectively, and Llama3-8B with approximately 8B parameters.

Table 2 compares the training step latency and speedup of TorchTitan-based configurations and EvoCompile under the same eager execution reference. EvoCompile achieves comparable or higher acceleration on most workloads. For example, on Qwen3 MoE with sequence length 1k and batch size 2, EvoCompile improves the speedup from 1.25× to 1.40× by reducing latency from 249 ms to 178 ms. On Qwen3 MoE with sequence length 2k and batch size 2, EvoCompile further improves the speedup from 1.31× to 1.38×.

These results demonstrate that EvoCompile can reduce the compilation-effectiveness gap in large-scale training scenarios and achieve optimization effectiveness comparable to carefully engineered training configurations.

4.4.3 Cross-backend Generalization

To evaluate whether the proposed optimization process can reduce the compilation-effectiveness gap across different compiler ecosystems, we apply EvoCompile to three compilation backends with their corresponding standard compilation interfaces. Specifically, we evaluate PyTorch Inductor with `torch.compile(model)`, OpenXLA with `torch_xla.compile(step)`, and Thunder with `thunder.jit(model)`.

Table 3 reports the additional acceleration achieved by EvoCompile over each backend-specific compilation baseline. Across all evaluated backends, EvoCompile consistently improves performance, achieving 1.27×–1.47× acceleration on Inductor, 1.23×–1.39× acceleration on OpenXLA, and 1.32×–4.34× acceleration on Thunder.

These results demonstrate that the compilation-effectiveness gap exists beyond a single compiler implementation. By adapting optimization decisions to workload characteristics, EvoCompile can further improve the utilization of different compiler backends.

Table 4: Ablation study of EvoCompile. Speedup denotes the geometric mean speedup relative to eager execution over each benchmark suite. Wins denotes the number of workloads that achieve further acceleration compared with naive compile.

Variant	HuggingFace		TIMM		TorchBench	
	Speedup	Wins	Speedup	Wins	Speedup	Wins
w/o Experience Memory	1.69	11	1.80	4	1.26	12
w/o Orchestrator	2.03	28	1.91	14	1.63	47
L1	2.14	30	1.99	17	1.74	53
L1 + L2	2.14	30	2.01	18	1.87	58
EvoCompile	**2.14**	**30**	**2.01**	**18**	**1.89**	**66**

4.5 Ablation Studies

We conduct ablation studies to investigate the contribution of major components in EvoCompile. Table 4 reports the geometric mean speedup relative to eager execution and the number of workloads that achieve additional acceleration over naive compile. The Wins metric evaluates whether each component helps EvoCompile discover further workload-specific optimization opportunities under the same optimization budget.

Effect of Experience Memory and Orchestrator.

We evaluate the contribution of experience-driven optimization components by removing the Experience Memory and Orchestrator modules separately. Under the same optimization budget, removing Experience Memory prevents EvoCompile from leveraging accumulated successful experiences and failure constraints, leading to substantially fewer additional optimization wins. For example, the number of additional wins on TorchBench decreases from 66 to 12, demonstrating that accumulated optimization knowledge is critical for improving optimization efficiency across workloads.

Removing the Orchestrator module also reduces the final optimization effectiveness under the same optimization budget. This indicates that adaptive refinement and exploration based on execution feedback are important for guiding the optimization process toward more effective workload-specific strategies.

Effect of Multi-level Intervention. We evaluate the contribution of different intervention levels by progressively adding intervention components, including L1, L1+L2, and the complete EvoCompile framework. The results show that additional intervention levels provide complementary optimization benefits. Compared with L1, introducing L2 further improves performance on workloads where program structures limit compiler optimization. The complete EvoCompile framework further introduces L3 kernel-level intervention, increasing the number of additional wins on TorchBench from 58 to 66. These results demonstrate that deeper intervention levels can expose optimization opportunities beyond compilation- and program-level adaptation.

Overall, the ablation results confirm that both experience-driven optimization and multi-level intervention contribute to reducing the compilation-effectiveness gap.

5 Conclusion and Limitation

We identify the compilation-effectiveness gap between general compiler usage and workload-specific optimization, and present EvoCompile, an experience-adaptive framework that reduces this gap by improving how workloads are presented to existing compilers. Extensive experiments demonstrate that EvoCompile consistently improves compilation effectiveness. The effectiveness of EvoCompile depends on the coverage of accumulated optimization experience, and unseen workload patterns may require additional exploration.

AI USE STATEMENT

In this work, we used generative AI tools only to aid and polish writing, including language refinement and translation assistance. We have not used generative AI tools for research ideation, literature retrieval and discovery, methodology design, experimental execution, result generation, mathematical proof, or drafting substantive sections of the paper. All AI-assisted writing was carefully reviewed and revised by the authors. The authors take full responsibility for the final content of this work, including all text, claims, and experimental results.

ETHICS STATEMENT

This work does not involve human subjects, private data, or sensitive information. The proposed framework focuses on improving deep learning compiler optimization efficiency and does not introduce additional risks related to privacy, safety, or fairness.

REPRODUCIBILITY STATEMENT

We provide detailed descriptions of the proposed framework, optimization pipeline, and experimental settings in the main paper and appendix. The evaluation uses publicly available benchmarks from the PyTorch ecosystem, including HuggingFace, TIMM, and TorchBench, with all benchmark configurations and implementation details documented. An anonymous code repository containing the implementation, experiment scripts, and configuration files is provided as supplementary material to facilitate reproduction of our results.

REFERENCES

Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. PyTorch 2: Faster machine learning through dynamic Python bytecode transforms and graph compilation. In *ASPLOS*, 2024.

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 *OSDI*, 2018a.

Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *NeurIPS*, 2018b.

Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Rozière, Jonas Gehring, Michal Czakon, and Hugh Leather. Large language models for compiler optimization, 2023.

Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Rozière, Jonas Gehring, Michal Czakon, and Hugh Leather. Meta large language model compiler: Foundation models of compiler optimization. arXiv:2407.02524, 2024.

Cursor Team. Cursor: Ai code editor. `https://cursor.com`, 2025.

Google Research. Xla: Optimizing compiler for machine learning. `https://www.tensorflow.org/xla`, 2020.

Zhihao Jia et al. Taso: Optimizing deep learning computation with automatic generation of graph substitutions. In *OSDI*, 2019.

Savini Kashmira, Jayanaka L. Dantanarayana, Thamirawaran Sathiyalogeswaran, Krisztian Flautner, Lingjia Tang, and Jason Mars. GraphMend: Code transformations for fixing graph breaks in PyTorch 2. arXiv:2509.16248, 2025.

Chris Lattner et al. Mlir: A compiler infrastructure for the end of moore's law. In *IEEE/ACM International Symposium on Code Generation and Optimization*, 2021.

Wanchao Liang, Tianyu Liu, Less Wright, Will Constable, Andrew Gu, Yining Huang, et al. Torchtitan: One-stop PyTorch native solution for production ready LLM pre-training. *arXiv preprint arXiv:2410.06511*, 2024.

NVIDIA. NVIDIA TensorRT. https://developer.nvidia.com/tensorrt. Accessed: 2026-09-26.

Anne Ouyang, Simon Guo, Simran Chen, et al. Kernelbench: Can LLMs write efficient GPU kernels? arXiv preprint, 2025.

Adam Paszke, Sam Gross, Francisco Massa, et al. Pytorch: An imperative style, high-performance deep learning library. https://pytorch.org, 2019.

Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. PET: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *OSDI*, 2021.

William Wen. Introduction to torch.compile. https://docs.pytorch.org/tutorials/intermediate/torch_compile_tutorial.html, 2023. PyTorch Tutorials.

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 *OSDI*, 2020.

A Extended Related Work

A.1 Deep Learning Compiler Optimization

Deep learning compiler systems improve execution efficiency through computation graph optimization, operator transformation, and hardware-aware code generation. XLA (Google Research, 2020) and TensorRT (NVIDIA) optimize deep learning workloads through graph-level transformations, operator fusion, and backend-specific optimizations. TVM (Chen et al., 2018a) provides a programmable compilation framework, while AutoTVM (Chen et al., 2018b) and Ansor (Zheng et al., 2020) automatically search efficient execution schedules for target hardware. Recent compiler infrastructures such as MLIR (Lattner et al., 2021) further improve compiler extensibility by enabling reusable intermediate representations and multi-level optimization pipelines. Beyond schedule optimization, TASO (Jia et al., 2019) and PET (Wang et al., 2021) explore semantics-preserving graph substitutions to identify more efficient computation graphs.

These systems primarily focus on improving optimization after computation has been represented in a compiler-friendly form. In contrast, EvoCompile studies how practical training workloads should be presented to existing compilers and automatically identifies workload-specific compilation strategies to reduce the compilation-effectiveness gap.

A.2 Workload Adaptation for Deep Learning Compilation

Modern compilation frameworks increasingly support dynamic deep learning programs while preserving flexible execution. PyTorch 2 (Ansel et al., 2024) combines TorchDynamo for graph capture, AOTAutograd for training graph construction, and TorchInductor for optimized code generation. Beyond compiler-internal mechanisms, recent work has explored adapting workloads to improve compilation effectiveness. GraphMend (Kashmira et al., 2025) applies source-level transformations to eliminate graph breaks caused by dynamic control flow and Python I/O, enabling larger regions to be captured by TorchDynamo. TorchTitan (Liang et al., 2024) incorporates regional compilation in large-scale training workflows, compiling selected computation regions rather than uniformly compiling the entire program.

These approaches demonstrate that workload presentation can substantially affect compilation effectiveness, but primarily address specific adaptation mechanisms or workload-specific compilation choices. In contrast, EvoCompile targets the broader compilation-effectiveness gap by automatically selecting and refining interventions across compilation, program, and kernel levels.

A.3 Experience-driven Compiler Optimization

Compiler optimization increasingly relies on search, learning, and execution feedback to reduce manual tuning. AutoTVM (Chen et al., 2018b) and Ansor (Zheng et al., 2020) search efficient

schedules within structured optimization spaces, while recent LLM-based approaches extend automated optimization to compiler and kernel generation tasks. LLMCompiler (Cummins et al., 2023; 2024) applies language models to compiler optimization, and KernelBench (Ouyang et al., 2025) studies LLM-generated GPU kernels.

These approaches primarily optimize individual programs or optimization decisions through search or generation. In contrast, EvoCompile accumulates reusable experience that associates workload characteristics with optimization interventions and execution outcomes, enabling future workloads to benefit from previously validated optimization knowledge.

B Implementation Details of Kernel-level Intervention

This section describes how EvoCompile implements kernel-level intervention in a compiler-compatible manner. The goal of kernel-level optimization is not to modify compiler internals or patch generated compiler outputs, but to provide additional optimization opportunities that can be incorporated by the existing compilation pipeline.

B.1 Recipe-based Kernel Integration

In EvoCompile, kernel-level optimization is represented as a closed optimization recipe operation rather than a direct modification to compiler-generated code. For example, an S6-fuse scheme can be instantiated through:

```
{"op": "replace_pattern", "kernel": "<id>"}
```

The operation is applied by the recipe executor before compilation begins. The kernel identifier can refer to either a predefined catalog kernel (e.g., `addmm_gelu`) or a generated kernel produced by TritonWriter. Unknown kernel identifiers are rejected to ensure that only validated optimization operations are executed.

This design keeps kernel optimization consistent with other EvoCompile interventions: the optimizer generates a structured recipe, while the execution framework materializes the required optimization artifact.

B.2 Kernel Registration Contract

Each kernel implementation follows a unified registration interface. The kernel loader resolves the corresponding implementation from the kernel catalog or generated kernel directory and invokes its registration function.

A valid kernel registration performs three operations:

(1) it installs an opaque operator through `torch.library`;

(2) it registers an FX transformation that rewrites matched computation patterns into the new operator;

(3) it avoids modifying compiler-generated files or internal compiler caches.

After registration, the compiler observes the new operator during the normal compilation process and can incorporate it into subsequent optimization.

For example, the catalog kernel `addmm_gelu` targets a missed GEMM epilogue fusion pattern. Instead of modifying generated Triton code, EvoCompile replaces the original operator sequence with a registered fused operator before compilation, allowing the compiler backend to optimize the updated graph.

B.3 Validation and Failure Handling

Kernel-level interventions are evaluated through the same execution and validation pipeline as other optimization recipes.

Table 5: Workloads evaluated on RTX 4090.

#	Workload	Dataset
1	XLNetLMHeadModel	HuggingFace
2	XGLMForCausalLM	HuggingFace
3	OPTForCausalLM	HuggingFace
4	MBartForCausalLM	HuggingFace
5	BartForCausalLM	HuggingFace
6	PLBartForCausalLM	HuggingFace
7	PegasusForCausalLM	HuggingFace
8	AlbertForMaskedLM	HuggingFace
9	pyhpc_isoneutral_mixing	TorchBench
10	vgg16	TorchBench
11	cm3leon_generate	TorchBench
12	demucs	TorchBench
13	tts_angular	TorchBench
14	pyhpc_turbulent_kinetic_energy	TorchBench
15	ghostnet_100	TIMM

The Collector and Evaluator record execution traces and kernel-level signals, including newly introduced custom operators. A successful fusion is identified through the appearance of the registered operator rather than by inspecting generated kernel source code.

Each kernel intervention is validated using two criteria: (1) numerical correctness compared with eager execution under the predefined tolerance; (2) measurable performance improvement over the frozen vanilla compilation baseline.

If a generated kernel fails during execution, the failure is handled by TritonWriter repair using the generated artifact and runtime traceback. The system does not restart retrieval because the optimization scheme remains valid and only the materialized implementation requires correction.

If a kernel executes correctly but does not improve performance, the trial is recorded as a negative optimization experience. Such outcomes update future retrieval and exploration decisions without invalidating the underlying optimization family.

Through this design, kernel-level optimization remains fully integrated with EvoCompile's experience-driven optimization loop while avoiding direct modification of compiler internals.

C Additional Experimental Results

C.1 Cross-hardware Generalization

To evaluate whether the optimization effectiveness of EvoCompile depends on a specific GPU platform, we conduct an additional evaluation on an NVIDIA RTX 4090 GPU. Unlike the main experiments conducted on NVIDIA A100 GPUs, this experiment examines whether EvoCompile can maintain optimization effectiveness under a different hardware environment.

We select 15 workloads from the PyTorch benchmark suites for evaluation on RTX 4090. The selected workloads are listed in Table 5. All workloads are evaluated under the same software configuration and correctness criteria as the main experiments.

Table 6 reports the performance comparison between EvoCompile and existing compilation strategies. The speedup is measured relative to eager execution under the RTX 4090 environment.

The results show that EvoCompile achieves consistent improvements over existing compilation strategies on RTX 4090. Compared with standard compilation configurations, EvoCompile further improves the average speedup from 1.29×–1.34× to 1.40×. These results provide additional evidence that the optimization capability of EvoCompile is not restricted to a single GPU platform.

Table 6: Performance comparison on RTX 4090 using 15 workloads. Speedup is measured relative to eager execution.

Method	Speedup
Naive Compile	1.29
Benchmark Compile(Default)	1.33
Reduce-Overhead	1.33
Max-Autotune	1.34
EvoCompile(GPT-5.6-Sol)	**1.40**