Compiler Optimization via LLM Reasoning for Efficient Model Serving

Sujun Tang, Christopher Priebe[*], Rohan Mahapatra[*], Lianhui Qin, Hadi Esmaeilzadeh
Alternative **C**omputing **T**echnologies (**ACT**) Lab
University of California San Diego

Abstract

While model serving has unlocked unprecedented capabilities, the high cost of serving large-scale models continues to be a significant barrier to widespread accessibility and rapid innovation. Compiler optimizations have long driven substantial performance improvements, but existing compilers struggle with neural workloads due to the exponentially large and highly interdependent space of possible transformations. Although existing stochastic search techniques can be effective, they are often sample-inefficient and fail to leverage the structural context underlying compilation decisions. We set out to investigate the research question of whether reasoning with large language models (LLMs), without any retraining, can leverage the context-aware decision space of compiler optimization to significantly improve sample efficiency. To that end, we introduce a novel compilation framework (dubbed **REASONING COMPILER**) that formulates optimization as a sequential, context-aware decision process, guided by a large language model and structured Monte Carlo tree search (MCTS). The LLM acts as a proposal mechanism, suggesting hardware-aware transformations that reflect the current program state and accumulated performance feedback. Monte Carlo tree search (MCTS) incorporates the LLM-generated proposals to balance exploration and exploitation, facilitating structured, context-sensitive traversal of the expansive compiler optimization space. By achieving substantial speedups with markedly fewer samples than leading neural compilers, our approach demonstrates the potential of LLM-guided reasoning to transform the landscape of compiler optimization. [1]

1 Introduction

The rise of model serving for LLMs, diffusion models, and other neural models has enabled a new class of intelligent systems, driving transformative applications in healthcare, education, and scientific discovery. These models incur significant computational demands during inference, which proportionally translate into substantial monetary costs. Driving down the cost of model serving is critical, not merely to broaden access and democratize inference, but to catalyze faster cycles of innovation in model design and deployment. Achieving this goal demands reducing inference runtime on computational infrastructure, resources that are not only expensive but also increasingly limited in availability. Compiler optimizations are a critical enabler, not only for cost-efficient inferencing across diverse applications but also for empowering rapid research iteration. According to Bill Dally [1], NVIDIA's Chief Scientist, compiler optimizations have driven over a 400$\times$ improvement in runtime over the past decade, compared to just 2.5$\times$ from semiconductor scaling.

Existing compilers struggle with neural models due to the exponentially large space of valid program transformations (e.g., tiling, fusion, and layout changes). Each decision, such as selecting a

[1]Code is available at https://github.com/Anna-Bele/LLM_MCTS_Search
[*]Equal contribution

tiling factor or a parallelization strategy, introduces dependencies and constraints that influence the feasibility and performance benefits of subsequent transformations. As a result, neural compilation has increasingly turned to stochastic search techniques, such as genetic algorithms and evolutionary strategies, for optimization [2–5]. While these methods have shown promise in discovering performant configurations, they are fundamentally sample-inefficient. As a result, these techniques often explore redundant or invalid configurations. They also overlook synergistic transformations that emerge only when decisions are made with contextual awareness.

We set out to investigate the research question of whether reasoning with large language models (LLMs), without any retraining, can leverage the context-aware decision space of compiler optimization to significantly improve sample efficiency. To that end, we introduce a novel compilation framework that couples large language model (LLM) reasoning with Monte Carlo tree search (MCTS) to guide compiler optimizations. Hence, in our approach, compiler optimization is cast as a sequential decision-making process, in which each transformation, such as tiling, fusion, or vectorization, is selected with awareness of the current program state, while assimilating downstream information and propagating its implications upstream to guide future decisions. Our approach avoids the prohibitive cost of fine-tuning LLMs as compilation policies and does not require additional training or task-specific adaptation. In this formulation, the LLM evaluates partial transformation sequences and proposes contextually appropriate next steps, drawing upon hardware-aware cost models and the historical trajectory of optimization decisions. The LLM serves as a context-aware proposal engine: given the current schedule and its observed performance, it generates candidate mutations that are likely to be effective in the context of the traversed trajectory. These LLM-guided reasoning choices are integrated into a Monte Carlo tree search (MCTS) framework, which provides a structured mechanism for balancing exploration and exploitation by evaluating LLM-suggested transformations, expanding promising branches, and leveraging rollout feedback to adaptively steer the search toward high-performing regions of the exponentially large optimization space.

This integration of LLM-based chain-of-thought (CoT) guidance with tree search combines contextual reasoning and adaptability with principled, structured decision-making, enabling the compiler to navigate the complexity of the optimization search space with significantly improved sample efficiency. Empirical results show that our method achieves up to 2.5× speedup over unoptimized code using just 36 program samples, whereas state-of-the-art black-box autotuners like TVM using Evolutionary Search require up to 16× more program samples to reach comparable performance. These results underscore the promise of LLM-guided reasoning in neural compilation for efficient and scalable model serving.

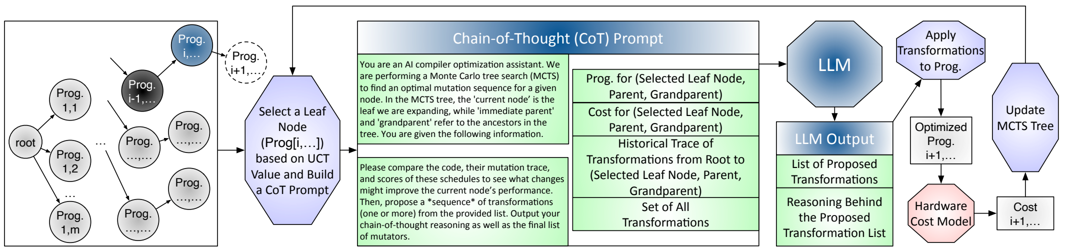
2 Problem Formalization

$$S_{\text{opt.}} = \underset{S' \subseteq M^*,\ |S'| \leq T}{\operatorname{argmax}} f\left((m'_k \circ \cdots \circ m'_1)(p_0)\right) \tag{1}$$

We consider the problem of optimizing an input program $p_0 \in P$ representing a layer from a neural network for some objective function $f : P \mapsto \mathbb{R} \geq 0$. This objective function represents an evaluation of the program on the target platform for some figure of merit (e.g., latency, power, utilization). Any program $p \in P$ can be transformed through the application of some transformation/optimization (used interchangeably from here on out) $m \in M$, where each optimization is a function $m : P \mapsto P$ that performs a targeted transformation to the program, thus introducing a new variant of the program that is semantically equivalent to the original program but may perform better or worse on a target hardware platform. In this way, successive application of transformations to a program can yield significant performance differences from the original. Therefore, given some maximum transformation sequence length T, the goal is to find a sequence of transformations $S_{\text{opt.}} = \langle m_1, m_2, \ldots, m_n \rangle$ such that $n \leq T$ and $f(p_{\text{opt.}}) = \max_{S' \subseteq M^*,\ |S'| \leq T} f\left((m'_k \circ \cdots \circ m'_1)(p_0)\right)$ where $p_{\text{opt.}} = (m_n \circ m_{n-1} \circ \cdots \circ m_1)(p_0)$ and M^* is the Kleene star[2] of M. These constraints collectively define the optimization objective given in Equation (1).

To facilitate an efficient search over the space of valid program transformation sequences, we cast the optimization problem as a finite-horizon Markov decision process (MDP) defined by the tuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$. This formulation provides a structured approach for sequential decision-

[2]The Kleene star operator, denoted with an asterisk (*), represents the set of all finite-length sequences, including the empty sequence, formed from elements of a given set.


root
Prog. 1,1
Prog. 1,2
Prog. 1,m
Prog. i-1,...
Prog. i,...
Prog. i+1,...
Prog. ...,...
Select a Leaf Node (Prog[i,...]) based on UCT Value and Build a CoT Prompt
Chain-of-Thought (CoT) Prompt
Prog. for (Selected Leaf Node, Parent, Grandparent)
Cost for (Selected Leaf Node, Parent, Grandparent)
Historical Trace of Transformations from Root to (Selected Leaf Node, Parent, Grandparent)
Set of All Transformations
LLM
LLM Output
List of Proposed Transformations
Reasoning Behind the Proposed Transformation List
Apply Transformations to Prog.
Optimized Prog. i+1,...
Hardware Cost Model
Cost i+1,...
Update MCTS Tree

You are an AI compiler optimization assistant. We are performing a Monte Carlo tree search (MCTS) to find an optimal mutation sequence for a given node. In the MCTS tree, the 'current node' is the leaf we are expanding, while 'immediate parent' and 'grandparent' refer to the ancestors in the tree. You are given the following information.

Please compare the code, their mutation trace, and scores of these schedules to see what changes might improve the current node's performance. Then, propose a *sequence* of transformations (one or more) from the provided list. Output your chain-of-thought reasoning as well as the final list of mutators.

Figure 1: Overview of the optimization workflow. The algorithm explores the tree to select a candidate node. At this node, the LLM is prompted with contextual information to generate a sequence of transformations, which are then applied to produce optimized code variants.

making in the transformation space, allowing the search process to account for how individual transformations compound over time to affect final program performance. Compared to unstructured methods such as exhaustive or purely stochastic search, which often require a large number of expensive program evaluations, casting the problem as an MDP enables more deliberate exploration, offering the potential for improved sample efficiency. Each state $s_t \in \mathcal{S}$ corresponds to a program $p_t \in P$ obtained by applying a sequence of transformations to the original program p_0, i.e., $s_t = p_t = (m_t \circ \cdots \circ m_1)(p_0)$. An action $a_t \in \mathcal{A}$ corresponds to selecting a transformation $m \in M$ to apply at step t, transitioning the current program to a new variant. Since the application of a transformation is deterministic, the transition function $\mathcal{P}(s_{t+1} \mid s_t, a_t)$ is 1 if $s_{t+1} = a_t(s_t)$ and 0 otherwise. The reward function is defined as the objective value caused by the optimization sequence, i.e., $\mathcal{R}(s_t, a_t) = f(a_t(p_t))$. By formulating the problem as an MDP, we enable the use of planning algorithms such as Monte Carlo tree search (MCTS) to explore program transformation sequences. The search ultimately yields an optimized transformation sequence $S'_{\text{opt.}}$ that approximately maximizes the objective function, as defined in Equation (1).

3 REASONING COMPILER: Integrating LLM-Guided Contextual Reasoning with Monte Carlo Tree Search

We present the **REASONING COMPILER**, a novel compilation framework that unifies the structured exploration capabilities of Monte Carlo tree search (MCTS) with the contextual, history-aware reasoning of large language models (LLMs). While MCTS provides a principled approach to exploring sequences of program transformations, compiler optimization introduces a unique challenge: the successive application of transformations can exhibit complex, non-local interactions that are difficult to capture through purely stochastic or myopic policies. To address this, we employ an LLM to model program transformation context, tracking which transformations have been applied, how they impact performance, and what directions remain promising. This contextualization is essential to enabling effective and sample-efficient search in compiler optimization.

Optimization interaction is complex, complicating optimization. Unlike tasks where actions are relatively independent, program transformations compose in subtle and complex ways. For example, the profitability of applying loop tiling may depend on the prior application of loop fusion or unrolling. Additionally, transformations can introduce new, unforeseen opportunities/constraints for future transformations. These dependencies make the space of valid and useful transformation sequences both combinatorially large and deeply contextual. While black-box optimization methods such as evolutionary search and some implementations of reinforcement learning have achieved notable success in compiler autotuning [2, 5–7], they often do not explicitly model the nuanced structural and temporal dependencies between transformations. This can limit their ability to generalize across contexts where optimization efficacy depends on intricate transformation histories. Even when guided by local reward signals, they may struggle to capture the interplay between past decisions and future opportunities, limiting their effectiveness in deeply contextual optimization landscapes. Our insight is that efficient search in this space requires an agent capable of reasoning over the transformation history, structural code changes, and observed performance dynamics to inform the next transformation step.

3.1 LLM-Guided Contextual Reasoning for Program Transformation Proposal

Contextual reasoning via LLMs. To address these challenges, the **Reasoning Compiler** leverages a large language model (LLM) as a contextual reasoning engine. The LLM is tasked with synthesizing program transformation sequence proposals that are not only syntactically valid but also informed by the full history and structure of the program. By prompting the LLM with a rich, structured representation of the current optimization state, we enable it to reason over the cumulative effects of prior transformations, analyze performance trends, and identify differential improvements over prior programs.

Figure 1 illustrates the optimization workflow. From the root program, the **Reasoning Compiler** traverses the tree by computing the UCT score [8], selecting a promising leaf node v_i with corresponding program p_i for expansion by balancing exploitation of high-reward paths and exploration of under-sampled branches based on visit statistics and node costs (see §3.2).

Prompt construction. At each expansion step in the search, the LLM receives a prompt that includes the source code and predicted performance cost for the current program p_i, its parent p_{i-1}, and its grandparent p_{i-2}. It also includes the ordered sequences of transformations that were applied to reach each of these program variants, denoted S_i, S_{i-1}, and S_{i-2}. Finally, the full set of available transformation operations M is included. Given this context, the LLM is explicitly instructed to:

1. Analyze the differences between program variants and their associated costs, identifying which transformations contributed to observed performance changes.
2. Reason about potential interactions between previously applied and candidate future transformations, including both synergistic and antagonistic effects.
3. Synthesize a new sequence of transformations that is justified in the context of the current program structure and transformation history.
4. Provide a rationale for the proposed sequence, referencing specific code features, transformation interactions, and performance considerations.

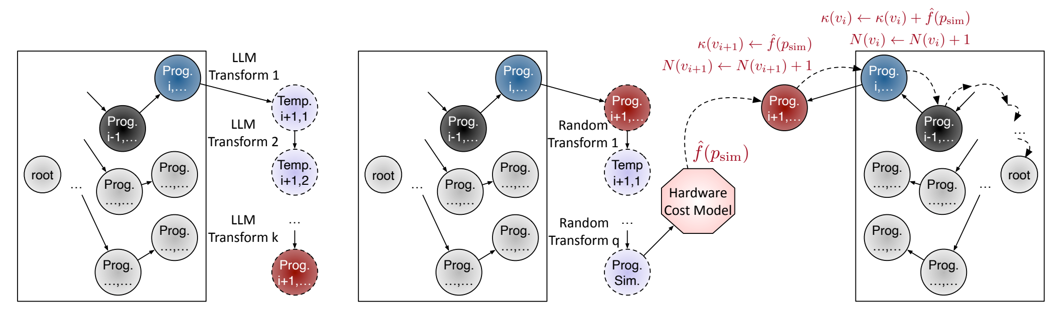
This structured prompt is designed to elicit chain-of-thought (CoT) reasoning [9], encouraging the LLM to perform deep, multi-step analysis and move beyond surface-level edits, instead generating proposals that are both semantically meaningful and tailored to the evolving optimization trajectory.

Transformation proposal and validation. The LLM proposes a candidate transformation sequence $S = \langle m_1, \dots, m_k \rangle$ in the form of a string containing transformation names. Given the generative nature of the LLM, the output may include invalid or unrecognized transformations even though it is guided by a predefined set of valid transformations To ensure correctness, the output string is first parsed and filtered to retain only those transformations that match known valid names and transformation parameters. If no valid transformations remain after this filtering, we fall back to sampling a single random transformation from the valid set. The successfully validated and applied sequence yields a new program variant p_{i+1}, with its transformation history updated as $S_{i+1} = S_i \oplus S$, where $\oplus$ denotes sequence concatenation. This new program variant is scored using a hardware cost model and used to update the MCTS tree (see §3.2).

It is important to emphasize that the LLM is not the centerpiece of our contribution, but a necessary enabler of effective search in this domain. Compiler optimization poses a uniquely challenging setting due to the non-local, compositional nature of transformation interactions. Traditional black-box search or heuristic-guided methods struggle to navigate such spaces efficiently. The **Reasoning Compiler** uses structured search (via MCTS) in conjunction with learned contextual reasoning (via LLM + CoT) to overcome these challenges. The result is a sample-efficient optimization algorithm capable of discovering performant transformation sequences in high-dimensional, high-interaction spaces.

3.2 Structured Optimization via Monte Carlo Tree Search

MCTS as a sample-efficient planner. As described in §2, we cast program optimization as a finite-horizon decision process over the space of transformation sequences. Framing the problem as an MDP allows the **Reasoning Compiler** to consider long-term transformation effects and invoke planning algorithms such as Monte Carlo tree search (MCTS) to explore this space deliberately and efficiently.



LLM Transform 1
Temp. i+1,1
LLM Transform 2
Temp. i+1,2
LLM Transform k
Prog. i+1,...
Prog. i,...
Prog. i-1,...
root
Random Transform 1
Temp i+1,1
Random Transform q
Prog. Sim.
Hardware Cost Model
$\hat{f}(p_{\text{sim}})$
$\kappa(v_{i+1}) \leftarrow \hat{f}(p_{\text{sim}})$
$N(v_{i+1}) \leftarrow N(v_{i+1}) + 1$
$\kappa(v_i) \leftarrow \kappa(v_i) + \hat{f}(p_{\text{sim}})$
$N(v_i) \leftarrow N(v_i) + 1$

(a) MCTS Expansion via LLM Mutations (b) MCTS Simulation (c) MCTS Backpropagation

Figure 2: Structured tree search where nodes are selected, expanded with LLM suggested transformations, simulated using hardware cost model for estimated costs, and updated with performance estimates to guide future search.

MCTS operates over a tree $\mathcal{T} = \langle V, E \rangle$ where each node $v_t \in V$ represents a program state $s_t \in \mathcal{S}$ corresponding to a program variant p_t, and each edge $e \in E$ corresponds to an action $a_t \in \mathcal{A}$ (i.e., a transformation sequence $S \in M^*$). This tree structure naturally supports the reuse of common transformation prefixes and allows the planner to backpropagate value estimates from downstream program variants to upstream decisions. Such reuse is critical in compiler optimization, where transformation sequences exhibit both compounding effects and long-range interactions.

Selection via UCT. During the selection phase, MCTS traverses $\mathcal{T}$ from the root, recursively selecting children v_i to maximize the UCT (Upper Confidence bounds applied to Trees) criterion:

$$\text{UCT}(v_i) = \frac{1}{\kappa(v_i)N(v_i)} + c\sqrt{\frac{\ln N(v_{i-1})}{N(v_i)}}$$

where $\kappa(v_i)$ is the estimated cost of v_i, $N(v_i)$ is the visit count of node v_i, and c governs the exploration-exploitation tradeoff.

LLM-guided expansion. As shown in Figure 2(a), once a promising leaf node v_i is selected, an LLM is queried to propose a transformation sequence conditioned on the program p_i at v_i and its ancestors (see §3.1). The model generates a candidate transformation sequence $S' = \langle m'_1, \ldots, m'_k \rangle$, which is applied to p_i to produce a new program $p_{i+1} = (m'_k \circ \cdots \circ m'_1)(p_i)$. This results in a new node v_{i+1} added to $\mathcal{T}$ corresponding to the updated program and extended transformation path. By leveraging the LLM's contextual reasoning, the system proposes globally informed transformations that extend beyond myopic heuristics.

Simulation for local cost estimation. To assess the likely long-term consequences of the LLM-suggested transformation sequence at v_{i+1}, the **Reasoning Compiler** simulates a short random trajectory starting from the new program p_{i+1}. This is done by sampling a randomized sequence of legal transformations $m_1, \ldots, m_q$ and applying them to obtain a terminal program $p_{\text{sim}} = (m_q \circ \cdots \circ m_1)(p_{i+1})$. This simulates the potential downstream development of the current program state, modeling how the compiler might continue to optimize from v_{i+1} onward under plausible future actions. The cost model then evaluates this trajectory to produce an estimated cost $\kappa(v_{i+1}) = \hat{f}(p_{\text{sim}})$. This simulation step provides a noisy but informative proxy for the downstream impact of reaching v_{i+1}, allowing MCTS to balance immediate and future consequences without expensive real-hardware runs.

As shown in Figure 2(b), once a new node v_{i+1} is added to the tree, the **Reasoning Compiler** performs a lightweight MCTS simulation by randomly walking down a path to estimate the expected utility that further optimizing v_{i+1} would attain in the future. This simulation is not a hardware simulation; rather, it is a common terminology used in MCTS algorithms. The performance of the program needs to be measured in these possible future nodes because the overall objective is to assess whether v_{i+1} will eventually minimize runtime. Recall that the overall objective is to minimize a true hardware-level cost function f (see §2). However, evaluating f directly would require compiling and running the program on real hardware, which is an expensive and often impractical operation within the inner loop of a planning algorithm. To address this, the **Reasoning Compiler** approximates

f using a learned hardware-aware cost model $\hat{f}$, which is trained to mimic hardware performance while remaining significantly cheaper to evaluate. This surrogate model is denoted $\hat{f}$.

Backpropagation. As illustrated in Figure 2(c), the estimated cost $\kappa(v_{i+1})$ is then backpropagated to all ancestors along the path to the root according to the update step $\kappa(v_A) \leftarrow \kappa(v_A) + \kappa(v_{i+1})$ where v_A is some ancestor. The visit counts are also updated according to the update step $N(v_A) \leftarrow N(v_A) + 1$. These updates refine the empirical estimates that guide future selections.

4 Results

We implement the **Reasoning Compiler** as an extension to MetaSchedule [5]. The framework introduces three modular components: (1) a prompt generator that serializes the current scheduling state, including the IRModule, transformation trace (i.e., the applied schedule history), and hardware cost model outputs, into structured prompts that capture the textual difference from the base IRModule and reflect the current schedule's performance; (2) an LLM interface that queries an external API (e.g., OpenAI) and parses LLM output into candidate transformation sequences; and (3) a tree manager that performs MCTS with selection based on UCT score, expansion using LLM suggested transformations, simulation with hardware-aware cost model, and backpropagation for tree statistics updates.

4.1 Experimental Setup

We evaluate **Reasoning Compiler** on four representative computational kernels from production-scale models: (1) Self-Attention layer from Llama3-8B [10], (2) Mixture-of-Experts (MoE) layer from DeepSeek-R1 [11], (3) Self-Attention layer from FLUX (Stable Diffusion) [12], and (4) Convolution layer from FLUX [12]. Compiler optimization is framed as a sequential decision process and guided by MCTS [13], using the Upper Confidence Bound for Trees (UCT) criterion [8] with exploration parameter $c = \sqrt{2}$ and a branching factor $B = 2$, following prior work [14, 15]. During search, the LLM (OpenAI GPT-4o mini [16]) is queried using hierarchical context—specifically, the parent and grandparent schedules and their transformations—to enable informed proposal generation. We compare three optimization strategies: (1) MetaSchedule [5], which uses Evolutionary Search; (2) MCTS without LLM involvement (MCTS); and (3) LLM+MCTS that uses prompt-based proposal generation (LLM-Guided MCTS). All experiments are conducted on a dedicated Intel Core i9 CPU using Apache TVM v0.20.0 [6] [17], with the hardware environment held constant to isolate the effects of scheduling decisions. Each experiment is repeated 20 times, and we report the mean performance to ensure statistical stability. We further eliminate noise by disabling background processes and ensuring no competing workloads during measurement. Additionally, we leverage OpenAI and HuggingFace model serving APIs to access the respective models. The implementation is open-sourced.

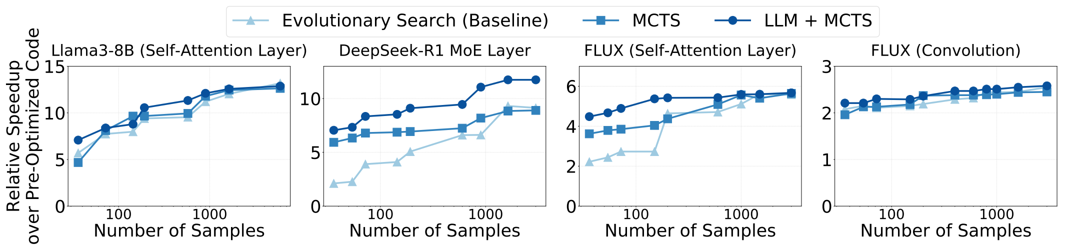


Figure 3: Relative speedup over pre-optimized code as a function of evaluated transformation proposals, illustrating the sample efficiency of LLM-guided compilation.

4.2 Evaluation

We assess the sample efficiency of our LLM-guided compilation framework by analyzing how code quality evolves with increasing search budget, quantified in terms of evaluated transformation proposals. Figure 3 and Figure 4 present results across four representative workloads, encompassing both transformer-style attention layers and convolution-heavy architectures. Across all benchmarks, our method achieves competitive or superior code performance with significantly fewer samples than state-of-the-art black-box autotuners such as MetaSchedule with Evolutionary Search. These results directly support the central hypothesis of our work: that leveraging LLM-driven, context-aware reasoning enables more efficient and effective exploration of the compiler optimization space.

Rapid convergence in low-sample regimes. A consistent trend across all benchmarks is the rapid ascent of code quality in the initial stages of search. This early-stage performance is critical in practice, as real-world compiler pipelines often operate under strict tuning time budgets. Figure 3 shows *Relative Speedup over Pre-Optimized Code* on the y-axis, with the number of evaluated transformation proposals on the x-axis. Speedup is defined as the ratio of the execution time of the unoptimized code to that of the optimized code after tuning. Higher values indicate more efficient and optimized codes. For instance, on the Llama3-8B Self-Attention Layer, the proposed method achieves a 7.08× speedup over the untuned baseline with just 36 samples, whereas Evolutionary Search requires 72 samples, which is twice the budget to achieve comparable gains. On the DeepSeek-R1 MoE Layer, the gap is even more pronounced: LLM-guided search achieves 7.05× speedup at 36 samples, while Evolutionary Search falls short of this mark even after 3000 samples.

Quantitative sample efficiency. To formally quantify sample efficiency, we compare the number of samples required by each method to reach target speedups. On the FLUX Self-Attention Layer, LLM-Guided MCTS attains a 2× speedup using only 36 samples, while Evolutionary Search requires more than 600 samples—a 16× reduction in tuning cost. On the FLUX Convolution Layer, our approach consistently outperforms Evolutionary Search across nearly all budget levels and reaches Evolutionary Search 's final performance after evaluating just 400 samples.

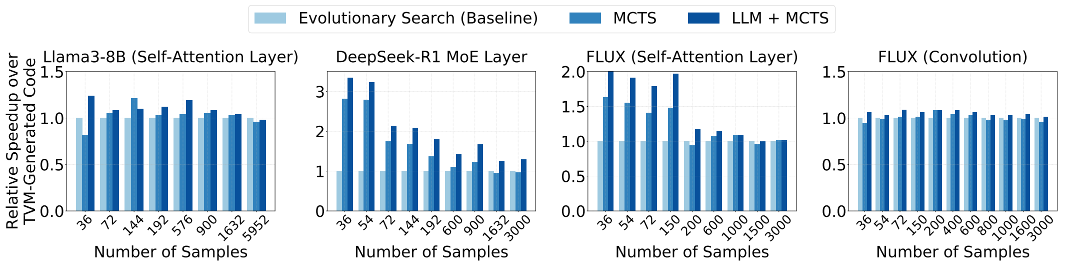


Figure 4: Relative speedup over Evolutionary Search-tuned code, emphasizing the remaining performance gap. LLM-Guided MCTS achieves superior efficiency, discovering high-quality codes with fewer samples across all operators in low-budget regimes.

Speedup relative to baselines. Figure 4 shows *Relative Speedup over Evolutionary Search-Generated Code*, again with number of samples on the x-axis. Here, speedup is the ratio between Evolutionary Search 's best code and ours at each point in the search. Values greater than 1 indicate superior performance relative to Evolutionary Search. These metrics highlight that LLM-Guided MCTS not only produces better codes, but does so *more aggressively and earlier* in the search process. For example, on the DeepSeek-R1 MoE Layer, LLM-Guided MCTS achieves a 3.3× speedup over Evolutionary Search at 36 samples, with the gap narrowing over time as Evolutionary Search eventually converges. This trend, which shows strong initial gains followed by convergence, demonstrates that LLM-Guided MCTS quickly identifies high-performing regions of the search space, while Evolutionary Search 's uninformed search requires substantial exploration to reach similar quality.

Operator-specific trends. We observe that certain operator types, such as matrix multiplication operations extracted from attention layers, exhibit sharper performance improvements. This is likely due to recurring structural patterns such as loop fusion, tiling, and vectorization, which pretrained LLMs can more readily recognize and exploit. Convolutional operators, by contrast, expose a broader and less regular transformation space. Nonetheless, our approach consistently matches or exceeds baseline performance with fewer samples, underscoring its effectiveness across diverse operator characteristics.

Implications. These findings reinforce our core thesis: compiler optimization should be cast as a structured decision process, enriched by prior knowledge and contextual reasoning. Our integration of LLMs into Monte Carlo tree search results in a strategically guided and sample-efficient search, particularly valuable in scenarios with constrained tuning budgets. By generating performant codes with orders-of-magnitude fewer samples, our framework offers both practical deployment advantages and a compelling alternative to conventional, sample-inefficient compilation pipelines.

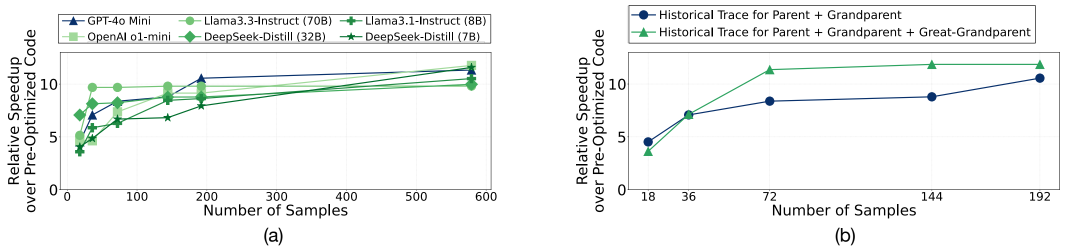


Figure 5: Ablation studies on LLM-Guided MCTS for the Llama3-8B Self-Attention Layer benchmark. (a) Comparing different LLMs as proposal engines shows that stronger LLMs lead to faster convergence. (b) Increasing the prompt's historical trace depth improves sample efficiency.

4.3 Ablation Study

4.3.1 Impact of LLM Choice and Reasoning Strategy

To better understand the contributions of different components in our approach, we conduct an ablation study focused on the effects of LLM selection and reasoning modality. Figure 5(a) shows the relative speedup over unoptimized code as a function of the number of schedule samples evaluated by LLM-Guided MCTS on the Llama3-8B Self-Attention Layer benchmark, using a range of LLM models for API calls. The x-axis indicates the cumulative number of schedules explored, while the y-axis shows the best speedup achieved so far. This setup enables us to directly compare how effectively various LLMs leverage contextual information to guide the search. These general trends of the results support our central claim: compiler optimization benefits from goal-directed, context-aware reasoning in terms of sample efficiency. Below, we discuss the specific behaviors that exemplify different reasoning strategies.

Large instruction-tuned Llama3.3 (70B) achieves exceptional sample efficiency. The instruction-tuned Llama3.3-70B model rapidly attains near-optimal performance, reaching a 9.69× speedup after only 36 samples—roughly 86% of the GPT-4o mini's maximum speedup but using less than 6% of its sampling budget. This corresponds to an approximately 15× improvement in sample efficiency. Instruction tuning also significantly improves the ability of LLMs to generate domain-specific, context-aware transformation proposals. The consistent performance advantage of instruction-tuned models over untuned counterparts of comparable size confirms that semantic task alignment, combined with sufficient model capacity, synergistically enhances the effectiveness of sequential context reasoning in guiding compiler optimizations.

DeepSeek-R1-Distill-Qwen (32B) excels in long-horizon optimization. The DeepSeek-R1-Distill-Qwen-32B model, employing a Mixture-of-Experts (MoE) architecture, exhibits a more gradual improvement, starting with a 7.07× speedup at 18 samples and reaching 9.98× after 579 samples. The sparse expert routing inherent in MoE architectures likely facilitates exploration of complex transformation sequences over extended horizons, complementing context-aware reasoning by enabling specialized and conditional decision-making.

Lower-parameter models also achieve high sample efficiency. Despite their reduced scale, smaller models still produce notable speedups relative to the untuned baseline. For example, Llama3.1-Instruct (8B) reaches a 5.87× speedup, and DeepSeek-R1-Distill-Qwen (7B) achieves 4.86× at just 36 samples. When compared to the widely used Evolutionary Search strategy, which requires around 72 samples to achieve a 7× speedup and fails to reach comparable performance for tuning the DeepSeek-R1 MoE Layer even after 3000 samples, these smaller models consistently outperform. LLM-Guided MCTS with lower-parameter models achieve at least twice the sample efficiency of Evolutionary Search, making them well-suited for efficient compiler optimization in local or edge deployments.

Open-source models match proprietary model performance. Our results demonstrate that open-source LLMs, when adequately scaled and instruction-tuned, match or exceed the performance of proprietary baselines such as GPT-4o mini. This underscores the broad applicability of our approach and its independence from proprietary data or architectures, enabling widespread adoption of context-aware, LLM-guided compiler optimization.

4.3.2 Impact of Historical Trace Depth on Optimization Efficiency

Figure 5(b) presents the relative speedup over unoptimized code as a function of the number of schedule samples evaluated by LLM-Guided MCTS on the Llama3-8B Self-Attention Layer benchmark. Using a deeper historical trace (see Figure 1) in the prompt (parent + grandparent + great-grandparent) leads to faster convergence compared to the shallower trace (parent + grandparent). For example, at 36 samples, the deeper trace achieves a speedup of approximately 7.13×, slightly surpassing the 7.08× of the shallower trace. However, by 72 samples, the deeper trace saturates at 11.36× speedup, while the shallower trace reaches only 8.38×, requiring many more samples (around 579) to approach 11.3× performance. This demonstrates that including longer historical context enables the LLM to better capture dependencies and synergies in transformation sequences, resulting in more sample-efficient and goal-directed exploration, validating the advantage of context-aware reasoning.

5 Related Work

ML-Based Auto-Tuning. Auto-tuning frameworks optimize performance-critical parameters (e.g., loop tile sizes, phase orderings, memory layouts) using a variety of ML-based techniques, including linear models [18, 19], tree-based methods [20, 21], Bayesian networks [22, 23], evolutionary algorithms [20, 24, 25], clustering [19, 25], and reinforcement learning [7, 24–26]. The REASONING COMPILER shares the same goal of performance-driven parameter selection, but distinguishes itself by combining LLM-based contextual reasoning with structured search (via MCTS) to explore transformation sequences in a history- and structure-aware manner.

Machine Learning for Neural Compilation. Machine learning has been extensively applied to optimize neural network inference pipelines, including high-level graph-level optimizations [27–34] and low-level code generation [3, 6, 35–40]. Systems such as TVM/Ansor [2, 4, 6] and FlexTensor [41] employ learned cost models and evolutionary strategies to navigate large configuration spaces. While these approaches are highly effective at tuning tensor programs, they typically focus on local parameter optimization or rely on domain-specific heuristics. The REASONING COMPILER moves beyond these works by introducing contextual reasoning through LLMs, enabling the system to reason over transformation history, structural changes, and performance trends, an approach not explored in prior neural compilation work.

LLMs for Code Reasoning and Optimization. LLMs have demonstrated capabilities in code generation [42–47], fuzzing [48], and bug repair [49]. Recent work has explored the use of LLMs to generate phase orderings or perform disassembly [50, 51]. The REASONING COMPILER advances these approaches by embedding an LLM in a structured decision loop, leveraging it for context-aware reasoning within a grounded search process.

6 Conclusion

The cost and complexity of compiling neural workloads remain major barriers to scalable and accessible model serving. Traditional compilers struggle with the vast, interdependent transformation space, while existing stochastic methods often lack sample efficiency and contextual awareness. We introduced REASONING COMPILER, a novel framework that formulates compiler optimization as a sequential, context-aware decision process. Using LLM as proposal generators and structured Monte Carlo Tree Search (MCTS) for guided exploration, REASONING COMPILER efficiently navigates the optimization space by combining LLM reasoning with performance feedback. Our results show that LLM-guided compilation, without any retraining, can substantially improve sample efficiency.

References

[1] William J. Dally. Hardware for deep learning. Keynote presented at Hot Chips 2023, Stanford University, August 29, 2023. Available via NVIDIA On-Demand at https://www.nvidia.com/en-us/on-demand/session/hotchips2023-keynote/, 2023.

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

[3] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv*, 2018.

[4] 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.

[5] 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 *NeurIPS*, 2022.

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *OSDI*, 2018.

[7] Byung Hoon Ahn, Prannoy Pilligundla, and Hadi Esmaeilzadeh. Chameleon: Adaptive code optimization for expedited deep neural network compilation. In *ICLR*, 2020.

[8] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *ECML*, 2006.

[9] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *NeurIPS*, 2022.

[10] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, et al. The Llama 3 Herd of Models. *arXiv*, 2024.

[11] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. *arXiv*, 2025.

[12] Black Forest Labs. Flux. Available at https://github.com/black-forest-labs/flux, 2024.

[13] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[14] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *CG*, 2007.

[15] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2–3):235–256, 2002.

[16] OpenAI. Openai GPT-4o mini API. Available at https://platform.openai.com/docs/models/gpt-4o-mini, 2025.

[17] Apache TVM Community. Apache TVM v0.20.0. Available at https://github.com/apache/tvm/releases/tag/v0.20.0, 2025.

[18] Mark Stephenson and Saman Amarasinghe. Predicting unroll factors using supervised classification. In *CGO*, 2005.

[19] Amir H. Ashouri, Andrea Bignoli, Gianluca Palermo, Cristina Silvano, Sameer Kulkarni, and John Cavazos. Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning. *ACM Trans. Archit. Code Optim.*, 14(3), 2017.

[20] Douglas Simon, John Cavazos, Christian Wimmer, and Sameer Kulkarni. Automatic construction of inlining heuristics using machine learning. In *CGO*, 2013.

[21] Ameer Haj-Ali, Hasan Genc, Qijing Huang, William Moses, John Wawrzynek, Krste Asanović, and Ion Stoica. Protuner: Tuning programs with monte carlo tree search. *arXiv*, 2020.

[22] Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, Eunjung Park, John Cavazos, and Cristina Silvano. Cobayn: Compiler autotuning framework using bayesian networks. *ACM Trans. Archit. Code Optim.*, 13(2), 2016.

[23] Erik Orm Hellsten, Artur Souza, Johannes Lenfers, Rubens Lacouture, Olivia Hsu, Adel Ejjeh, Fredrik Kjolstad, Michel Steuwer, Kunle Olukotun, and Luigi Nardi. Baco: A fast and portable bayesian compiler optimization framework. In *ASPLOS*, 2023.

[24] Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and David Li. Mlgo: a machine learning guided compiler optimizations framework. *arXiv*, 2021.

[25] Haolin Pan, Yuanyu Wei, Mingjie Xing, Yanjun Wu, and Chen Zhao. Towards efficient compiler auto-tuning: Leveraging synergistic search spaces. In *CGO*, 2025.

[26] Ameer Haj-Ali, Qijing Jenny Huang, John Xiang, William Moses, Krste Asanovic, John Wawrzynek, and Ion Stoica. Autophase: Juggling hls phase orderings in random forests with deep reinforcement learning. In *MLSys*, 2020.

[27] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. Deep learning with dynamic computation graphs. *arXiv*, 2017.

[28] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: Optimizing deep learning computation with automatic generation of graph substitutions. In *SOSP*, 2019.

[29] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. Optimizing cnn model inference on cpus. In *ATC*, 2019.

[30] Yaoyao Ding, Ligeng Zhu, Zhihao Jia, Gennady Pekhimenko, and Song Han. Ios: Inter-operator scheduler for cnn acceleration. In *MLSys*, 2021.

[31] Yanqi Zhou, Sudip Roy, Amirali Abdolrashidi, Daniel Wong, Peter Ma, Qiumin Xu, Hanxiao Liu, Mangpo Phitchaya Phothilimtha, Shen Wang, Anna Goldie, Azalia Mirhoseini, and James Laudon. Transferable graph optimizers for ml compilers. In *NeurIPS*, 2020.

[32] Zhen Zheng, Pengzhan Zhao, Guoping Long, Feiwen Zhu, Kai Zhu, Wenyi Zhao, Lansong Diao, Jun Yang, and Wei Lin. Fusionstitching: Boosting memory intensive computations for deep learning workloads. *arXiv*, 2021.

[33] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. Equality saturation for tensor graph superoptimization. In *MLSys*, 2021.

[34] Jie Zhao, Xiong Gao, Ruijie Xia, Zhaochuang Zhang, Deshi Chen, Lei Chen, Renwei Zhang, Zhen Geng, Bin Cheng, and Xuefeng Jin. Apollo: Automatic partition-based operator fusion through layer by layer optimization. In *MLSys*, 2022.

[35] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *CGO*, 2019.

[36] Jian Weng, Animesh Jain, Jie Wang, Leyuan Wang, Yida Wang, and Tony Nowatzki. Unit: Unifying tensorized instruction compilation. In *CGO*, 2021.

[37] Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodik, and Vinod Grover. Fireiron: A data-movement-aware scheduling language for gpus. In *PACT*, 2020.

[38] Yaoyao Ding, Cody Hao Yu, Bojian Zheng, Yizhi Liu, Yida Wang, and Gennady Pekhimenko. Hidet: Task-mapping programming paradigm for deep learning tensor programs. In *ASPLOS*, 2023.

[39] Rui Li, Yufan Xu, Aravind Sukumaran-Rajam, Atanas Rountev, and P. Sadayappan. Analytical characterization and design space exploration for optimization of cnns. In *ASPLOS*, 2021.

[40] Wookeun Jung, Thanh Tuan Dao, and Jaejin Lee. Deepcuts: A deep learning optimization framework for versatile gpu workloads. In *PLDI*, 2021.

[41] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *ASPLOS*, 2020.

[42] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv*, 2023.

[43] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. Starcoder 2 and the stack v2: The next generation. *arXiv*, 2024.

[44] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Empowering code generation with oss-instruct. *arXiv*, 2023.

[45] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *KDD*, 2023.

[46] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. Codet5+: Open code large language models for code understanding and generation. *arXiv*, 2023.

[47] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv*, 2024.

[48] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Universal fuzzing via large language models. *arXiv*, 2023.

[49] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *ICSE*, 2023.

[50] Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Kim Hazelwood, Gabriel Synnaeve, and Hugh Leather. Large language models for compiler optimization. *arXiv*, 2023.

[51] Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. Llm compiler: Foundation language models for compiler optimization. In *CC*, 2025.

A LLM Prompt Example

Below we show an example prompt used in our LLM-Guided MCTS framework (refer to Figure 1).

Example Code To be Optimized:

```
@tvm.script.ir_module
class MyModule:
   @T.prim_func
   def main(
      A: T.Buffer((1, 16, 7168), "float32"),
      B: T.Buffer((7168, 2048), "float32"),
      C: T.Buffer((1, 16, 2048), "float32"),
      ):
      T.func_attr({"global_symbol": "main", "tir.noalias": True})
      for b, t, j, k in T.grid(1, 16, 2048, 7168):
         with T.block("moe"):
            vb, vt, vj, vk = T.axis.remap("SSSR", [b, t, j, k])
            with T.init():
               C[vb, vt, vj] = T.float32(0)
            C[vb, vt, vj] += A[vb, vt, vk] * B[vk, vj]
```

Prompt:

```
You are an code optimization assistant performing Monte Carlo Tree
Search (MCTS) on a given code to improve performance. Each code
has a corresponding history of transformations and predicted cost.

You are given the code of the selected node and its immediate
parent. Their main differences include:

Loop shapes:
Current:
for b_1, t_1, j_1 in T.grid(1, 2, 8), and then
j_2_init in T.grid(1), and later
j_2 in T.grid(1).
Index example: vj = ... + j_1 * 64 + j_2 * 64
Parent:
for b_1, t_1, j_1 in T.grid(1, 2, 2),
then j_2_init in T.grid(4),
and later j_2 in T.grid(4).
Index example: vj = ... + j_1 * 256 + j_2 * 64

Tile decisions:
Current: sample_perfect_tile(..., decision=[4, 8, 1, 64])
Parent: sample_perfect_tile(..., decision=[4, 2, 4, 64])

Performance estimates:
Current: 0.773
Parent: 0.313

Available transformations:
TileSize, Parallel, ComputeLocation, Unroll

Task
Analyze the IR, trace, and predicted scores.
Then propose a sequence of transformations (you may repeat any) to
potentially improve performance.
Output the final suggested transformations in a single line
starting with "Transformations to apply:..."
```

```
For example:
Transformations to apply: TileSize, Unroll, ...
```

LLM Response:

```
Transformations to apply: TileSize, TileSize, Unroll, Parallel,
TileSize
```

B Sample Efficiency and Speedup Results

Table 1 presents the relative speedup of three methods—Evolutionary Search, MCTS, and LLM-Guided MCTS —evaluated across the different benchmarks. Speedup is measured as the ratio of execution time for the unoptimized code to that of the optimized code after applying a given number of transformation proposals. The table captures performance as a function of the number of samples explored. Higher values indicate more effective optimization. For instance, LLM-Guided MCTS consistently achieves higher speedups with fewer samples, demonstrating superior sample efficiency and faster convergence compared to MCTS and Evolutionary Search.

Table 1: Speedup over unoptimized code across varying numbers of samples for different compiler optimization methods.

	Number of Samples	18	36	72	144	192	600	900	1632	5952
Llama3-8B (Self-Attention Layer)	Evolutionary Search	4.67	5.70	7.74	7.98	9.40	9.54	11.20	12.04	13.18
	MCTS	4.14	4.68	8.11	9.65	9.66	9.94	11.79	12.44	12.63
	MCTS + LLM	4.52	7.08	8.38	8.79	10.56	11.33	12.10	12.57	12.87
	Number of Samples	36	54	72	144	192	600	900	1632	3000
DeepSeek-R1 (MoE Layer)	Evolutionary Search	2.11	2.27	3.90	4.10	5.07	6.60	6.62	9.31	9.13
	MCTS	5.93	6.33	6.79	6.87	6.93	7.24	8.18	8.84	8.90
	MCTS + LLM	7.05	7.33	8.34	8.53	9.10	9.45	11.06	11.74	11.74
	Number of Samples	36	54	72	150	200	600	1000	1500	3000
FLUX (Self-Attention Layer)	Evolutionary Search	2.22	2.44	2.73	2.73	4.64	4.71	5.11	5.61	5.58
	MCTS	3.62	3.79	3.85	4.04	4.37	5.09	5.56	5.40	5.64
	MCTS + LLM	4.48	4.67	4.89	5.37	5.42	5.43	5.59	5.60	5.67
	Number of Samples	36	54	72	150	200	400	600	800	1000
FLUX (Convolution)	Evolutionary Search	2.08	2.15	2.11	2.15	2.19	2.29	2.32	2.44	2.44
	MCTS	1.96	2.13	2.13	2.18	2.37	2.38	2.38	2.39	2.40
	MCTS + LLM	2.21	2.21	2.30	2.29	2.36	2.47	2.47	2.51	2.51

C Impact of LLM Choice and Reasoning Strategy

As a continuation of Figure 5(a), Table 2 reports speedup over unoptimized code on three additional benchmarks: DeepSeek-R1 MoE Layer, FLUX Self-Attention Layer, and FLUX Convolution Layer. Each block of the table corresponds to a different benchmark and shows the best speedup achieved by LLM-Guided MCTS as a function of the number of schedules sampled using the reasoning model listed in the table. Rows compare different reasoning models used for API call generation, including both proprietary (e.g., GPT-4o mini, OpenAI o1-mini) and open-source models (e.g., Llama3.3-Instruct, DeepSeek-Distill). Across all benchmarks, the results show that more capable models—those that are larger or instruction-tuned—consistently achieve higher speedups with fewer samples. For example, Llama3.3-Instruct (70B) and DeepSeek-Distill (32B) achieve near-maximal speedup within the first 72–150 samples, while smaller models such as DeepSeek-Distill (7B) or Llama3.1-Instruct (8B) reach similar performance more gradually. These results validate the generality of our findings: the use of context-aware LLMs accelerates convergence in LLM-Guided MCTS across diverse code domains. Moreover, the performance of open-source models is competitive with proprietary alternatives, further supporting the accessibility and reproducibility of our method.

Table 2: Speedup over unoptimized code across varying numbers of samples for different choices of API call models.

	Number of Samples	18	36	72	150	200	600
Llama3-8B (Self-Attention Layer)	GPT-4o mini	4.52	7.08	8.38	8.79	10.56	11.33
	OpenAI o1-mini	4.63	4.64	7.37	9.14	9.15	11.77
	Llama3.3-Instruct (70B)	5.15	9.68	9.69	9.80	9.80	9.81
	DeepSeek-Distill-Qwen (32B)	7.07	8.14	8.23	8.77	8.78	9.98
	Llama3.1-Instruct (8B)	3.60	5.87	6.28	8.46	8.63	10.52
	DeepSeek-Distill-Qwen (7B)	4.06	4.86	6.68	6.82	7.94	11.58
	Number of Samples	18	36	72	150	200	600
DeepSeek-R1 (MoE Layer)	GPT-4o mini	6.14	7.05	8.33	8.53	9.10	9.45
	OpenAI o1-mini	4.56	6.65	8.59	9.29	10.55	11.56
	Llama3.3-Instruct (70B)	7.30	7.70	7.96	8.06	8.60	9.22
	DeepSeek-Distill-Qwen (32B)	5.56	8.11	9.49	10.17	11.02	12.02
	Llama3.1-Instruct (8B)	4.29	4.31	6.98	8.70	9.18	9.21
	DeepSeek-Distill-Qwen (7B)	6.89	7.35	7.35	10.22	10.34	10.44
	Number of Samples	18	36	72	150	200	600
FLUX (Self-Attention Layer)	GPT-4o mini	4.09	4.48	4.89	5.37	5.42	5.43
	OpenAI o1-mini	3.29	2.99	5.27	5.53	5.65	5.67
	Llama3.3-Instruct (70B)	2.67	3.12	4.82	4.86	5.71	5.71
	DeepSeek-Distill-Qwen (32B)	3.56	4.29	4.29	4.54	4.99	5.21
	Llama3.1-Instruct (8B)	2.01	3.43	3.55	3.80	3.87	5.21
	DeepSeek-Distill-Qwen (7B)	3.02	3.76	3.83	4.54	4.94	5.17
	Number of Samples	18	36	72	150	200	600
FLUX (Convolution)	GPT-4o mini	1.65	2.21	2.30	2.29	2.36	2.47
	OpenAI o1-mini	2.37	2.37	2.38	2.39	2.45	2.54
	Llama3.3-Instruct (70B)	2.30	2.35	2.47	2.51	2.56	2.57
	DeepSeek-Distill-Qwen (32B)	1.41	2.26	2.32	2.35	2.40	2.45
	Llama3.1-Instruct (8B)	2.11	2.30	2.39	2.55	2.55	2.56
	DeepSeek-Distill-Qwen (7B)	1.56	2.18	2.42	2.44	2.46	2.45

D Impact of Historical Trace Depth on Optimization Efficiency

As a continuation of Figure 5(b), Table 3 presents the data for the ablation study on the depth of historical trace included in the prompt sent to the LLM. Specifically, we compare two configurations: the "Parent + Grandparent" setting, where the prompt contains information from the current node and its two immediate ancestors, and the "Parent + Grandparent + Great-Grandparent" setting, where the prompt additionally includes the great-grandparent node. These variations allow us to assess the impact of deeper context windows on the effectiveness of LLM-Guided MCTS.

Results show that increasing the historical context generally improves sample efficiency across all benchmarks. For example, on DeepSeek-R1 MoE Layer, adding one more ancestral node boosts early performance significantly, achieving a 9.39× speedup at just 18 samples compared to 6.14× for the shallower context. Similarly, on Llama3-8B Self-Attention Layer, the extended context leads to a higher final speedup (11.87× vs. 11.33×) and earlier convergence. The performance gains, while smaller, are also consistent on FLUX Self-Attention Layer and FLUX Convolution Layer, with improvements observed across all sample budgets. These findings confirm that providing richer historical context enables the LLM to make more informed decisions at each step of the search, ultimately enhancing the sample efficiency of LLM-Guided MCTS.

Table 3: Speedup over unoptimized code across varying numbers of samples for different context lengths.

Llama3-8B (Self-Attention Layer)	Number of Samples	18	36	72	150	200	600
	Parent + Grandparent	4.52	7.08	8.38	8.79	10.56	11.33
	Parent + Grandparent + Great-Grandparent	3.63	7.13	11.36	11.86	11.86	11.87
DeepSeek-R1 (MoE Layer)	Number of Samples	18	36	72	150	200	600
	Parent + Grandparent	6.14	7.05	8.33	8.53	9.10	9.45
	Parent + Grandparent + Great-Grandparent	9.39	10.31	10.31	10.49	10.59	10.65
FLUX (Self-Attention Layer)	Number of Samples	18	36	72	150	200	600
	Parent + Grandparent	4.09	4.48	4.89	5.37	5.42	5.43
	Parent + Grandparent + Great-Grandparent	4.21	4.55	4.81	5.47	5.53	5.61
FLUX (Convolution)	Number of Samples	18	36	72	150	200	600
	Parent + Grandparent	1.65	2.21	2.30	2.29	2.36	2.47
	Parent + Grandparent + Great-Grandparent	1.73	2.22	2.32	2.35	2.49	2.50

E Ablations of MCTS Branching Factor

To determine the value of MCTS branching factor (B), we ablate on $B = 2$ and $B = 4$. In Table 4, results show that when branching factor $B = 2$, LLM-Guided MCTS is more sample-efficient than when $B = 4$. Our choice of $B = 2$ aligns with prior works [14, 15]. If a higher branching factor is chosen, then there are more possible next steps, which require more sampling effort (i.e., more simulations) to cover these expanded possibilities at the same level of thoroughness.

Table 4: Speedup over unoptimized code across varying numbers of samples for different branching factors.

	Number of Samples	18	36	72	150	200	600
Llama3-8B (Self-Attention Layer)	B = 2	4.52	7.08	8.38	8.79	10.56	11.33
	B = 4	4.16	7.88	8.35	8.89	9.86	10.99
	Number of Samples	18	36	72	150	200	600
DeepSeek-R1 (MoE Layer)	B = 2	6.14	7.05	8.33	8.53	9.10	9.45
	B = 4	2.98	4.29	4.29	7.28	7.29	9.10
	Number of Samples	18	36	72	150	200	600
FLUX (Self-Attention Layer)	B = 2	4.09	4.48	4.89	5.37	5.42	5.43
	B = 4	2.40	3.48	3.97	4.95	4.97	5.55
	Number of Samples	18	36	72	150	200	600
FLUX (Convolution)	B = 2	1.65	2.21	2.30	2.29	2.36	2.47
	B = 4	1.91	1.97	2.23	2.23	2.25	2.43