
REASONING COMPILER: LLM-Guided Optimizations for Efficient Model Serving

Sujun Tang
 University of California
 San Diego
 sujun@ucsd.edu

Christopher Priebe*
 University of California
 San Diego
 cpriebe@ucsd.edu

Rohan Mahapatra*
 University of California
 San Diego
 rohan@ucsd.edu

Lianhui Qin
 University of California
 San Diego
 lianhui@ucsd.edu

Hadi Esmaeilzadeh
 University of California
 San Diego
 hadi@ucsd.edu

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 optimizations 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-informed transformations that reflect the current program state and accumulated performance feedback. 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.

¹Code is available at https://github.com/Anna-Bele/LLM_MCTS_Search

*Equal contribution

39th Conference on Neural Information Processing Systems (NeurIPS 2025).

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 tiling factor or a parallelization strategy, introduces dependencies and constraints that influence the feasibility and performance benefits of subsequent transformations. Rule-based optimizations also often rely on hand-tuned heuristics that can overfit to a specific workload or hardware target. The seminal work in superoptimization [1–3] aimed to tackle these shortcomings through enumerative or symbolic search, but the search space proved combinatorial and rugged. STOKE [4] showed that high-quality programs often lie in regions separated by low-probability paths, and therefore adopted Markov chain Monte Carlo (MCMC)-based randomized search. Neural compilers followed suit, using evolutionary search or simulated annealing to navigate similarly irregular landscapes [5–8]. While these methods have shown promise in discovering performant configurations, they are fundamentally sample-inefficient. They overlook synergistic transformations that emerge only when decisions are made with contextual awareness. These techniques also often explore redundant subspaces or invalid configurations.

In contrast, 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 optimizations to significantly improve sample efficiency. To that end, we introduce a novel compilation framework that couples *LLM reasoning with Monte Carlo tree search (MCTS)* to guide compiler optimization. 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 also *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, nor does it 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-informed cost models and the historical trajectory of optimization decisions to inform its proposal. The LLM serves as a context-aware proposal engine: given the current schedule and its observed performance, it generates candidate transformations that are likely to be effective in the context of the traversed trajectory. These LLM-guided reasoning choices are integrated into an MCTS framework that 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 search space with significantly improved sample efficiency. We evaluate **REASONING COMPILER** and compare its improvements and sample efficiency with TVM, which employs evolutionary search. Results show that **REASONING COMPILER** consistently achieves significantly higher speedups than what TVM achieves using significantly fewer samples. On five representative benchmarks (Llama-3-8B Attention Layer, DeepSeek-R1 MoE Layer, FLUX Attention Layer, FLUX Convolution Layer, and Llama-4-Scout MLP Layer) and across five hardware platforms (Amazon Graviton2, AMD EPYC 7R13, Apple M2 Pro, Intel Core i9, and Intel Xeon E3), **REASONING COMPILER** achieves $5.0\times$ average speedup using $5.8\times$ fewer samples, resulting in an average of $10.8\times$ improvement over TVM in sample efficiency. For the end-to-end Llama-3-8B benchmark across five hardware platforms, **REASONING COMPILER** uses $3.9\times$ fewer samples to achieve a $4.0\times$ speedup, yielding a $5.6\times$ sample efficiency improvement. These results underscore the promise of LLM-guided reasoning in neural compilation for efficient and scalable model serving.

2 Problem Formalization

$$S_{\text{opt.}} = \underset{S' \subseteq O^*, |S'| \leq T}{\operatorname{argmax}} f((o'_k \circ \dots \circ o'_1)(p_0)) \quad (1)$$

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) $o \in O$, where each optimization is a function $o : P \mapsto P$ 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 $S_{\text{opt.}} = \langle o_1, o_2, \dots, o_n \rangle$ such that $n \leq T$ and $f(p_{\text{opt.}}) = \max_{S' \subseteq O^*, |S'| \leq T} f((o'_k \circ \dots \circ o'_1)(p_0))$ where $p_{\text{opt.}} = (o_n \circ o_{n-1} \circ \dots \circ o_1)(p_0)$ and O^* is the Kleene star² of O . 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-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., $s_t = p_t = (o_t \circ \dots \circ o_1)(p_0)$. An action $a_t \in \mathcal{A}$ corresponds to selecting a transformation $o \in O$ 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. Under standard assumptions, such as finite branching, bounded rewards, and a tree policy (e.g., UCT) that guarantees persistent exploration, MCTS is consistent on finite-horizon problems: as the number of simulations tends to infinity, it converges (with probability 1) to the optimal root action/sequence $S_{\text{opt.}}$ that maximizes the objective. With any finite simulation budget, it returns a high-quality but approximate solution. Consequently, our framework (see §3) yields a sequence $S'_{\text{opt.}}$ that approximately maximizes the objective in practice while enjoying asymptotic optimality in theory.

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

We present **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 interactions are complex, making efficient search challenging. 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 combinatorial 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 [6, 8–10], they often do not explicitly model the nuanced structural and temporal dependencies between transformations. This can limit their ability to generalize across contexts, as optimization efficacy depends on 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

²The 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.

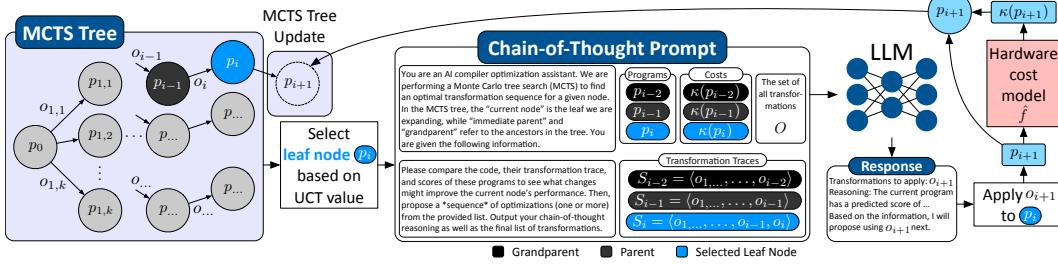


Figure 1: Overview of the optimization workflow. The algorithm explores the tree to select a candidate node. At this node, the LLM is prompted with contextual information to generate a sequence of transformations, which are then applied to produce optimized code variants.

is that efficient search in this space benefits from an agent that reasons over transformation history, structural code changes, and observed performance dynamics to choose the next step.

3.1 LLM-Guided Contextual Reasoning for Program Transformation Proposal

Contextual reasoning via LLMs. To address these challenges, **REASONING COMPILER** leverages a large language model (LLM) as a contextual reasoning engine. The LLM is tasked with synthesizing program transformation sequences 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, **REASONING COMPILER** traverses the tree by computing the UCT score [11], selecting a promising leaf node (i.e., 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 O 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; and (4) provide a rationale for the proposed sequence, referencing specific code features and transformation interactions. This structured prompt is designed to elicit chain-of-thought (CoT) reasoning [12], 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 $o_{i+1} \in O$ in the form of a string. Given the generative nature of the LLM, the output may include an invalid or unrecognized transformation 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 a transformation that matches known valid names and transformation parameters. If no valid transformation is found, **REASONING COMPILER** samples a random transformation from the valid set. The successfully validated and applied transformation yields a new program variant p_{i+1} , with its transformation history updated as $S_{i+1} = S_i \oplus \langle o_{i+1} \rangle$, 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. **REASONING COMPILER** uses structured search (via MCTS) with learned contextual reasoning (via LLM + CoT)*

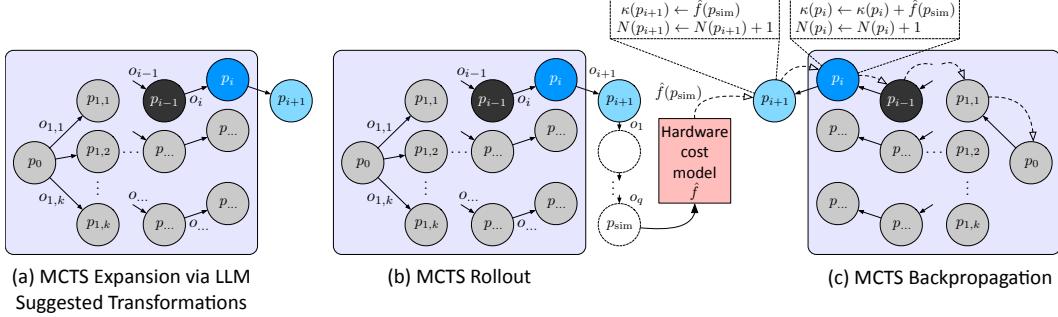


Figure 2: Structured tree search where nodes are (a) selected and expanded with the LLM suggested transformations, (b) scored by a learned hardware cost model, and (c) updated with performance estimates to guide future search.

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 **REASONING COMPILER** to consider long-term optimization effects and leverage planning algorithms such as Monte Carlo tree search (MCTS) to explore this space deliberately and efficiently.

MCTS operates over a tree $\mathcal{T} = \langle V, E \rangle$ where $V = P$ and $E = O$ such that each node $p \in P$ is a program from the state space S and each edge $o \in O$ corresponds to a transformation from the action space A . 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 child programs p_i to maximize the UCT (Upper Confidence bounds applied to Trees) criterion:

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

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

LLM-guided expansion. As shown in Figure 2(a), once a promising leaf node p_i is selected, an LLM is queried to propose a transformation conditioned on p_i and its ancestors (see §3.1). The model generates a candidate transformation $o_{i+1} \in O$, which is applied to p_i to produce a new program $p_{i+1} = o_{i+1}(p_i)$. This results in a new node p_{i+1} added to \mathcal{T} corresponding to the updated program and extended transformation path. To ensure \mathcal{T} remains acyclic, if p_{i+1} already exists in the tree, it is not added. By leveraging the LLM’s contextual reasoning, the system proposes globally informed transformations that extend beyond myopic heuristics.

Rollout for local cost estimation. As shown in Figure 2(b), once a new node p_{i+1} is added to the tree, **REASONING COMPILER** performs a lightweight MCTS rollout to estimate the long-term impact of the transformation sequence that produced it. This is done by sampling a randomized sequence of legal transformations o_1, \dots, o_q and applying them to obtain a terminal program $p_{\text{sim}} = (o_q \circ \dots \circ o_1)(p_{i+1})$. The objective is to minimize a hardware-level cost f (see §2), but evaluating f requires compiling and running on real hardware, which is too expensive for the inner loop of a planning algorithm. Following standard practice in compiler autotuning, **REASONING COMPILER** replaces f with a learned, hardware-informed surrogate \hat{f} that is cheap to evaluate and has been shown to accelerate search while preserving final quality [6, 7, 9, 13–15]. Therefore, the cost model evaluates this trajectory to produce an estimated cost $\kappa(p_{i+1}) = \hat{f}(p_{\text{sim}})$. This noisy but informative proxy captures the downstream impact of reaching p_{i+1} , enabling MCTS to balance immediate and future consequences without incurring real-hardware runs.

Backpropagation. As shown 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 $\kappa(p_A) \leftarrow \kappa(p_A) + \kappa(p_{i+1})$ where p_A is some ancestor program. The visit counts are also updated according to the update step $N(p_A) \leftarrow N(p_A) + 1$. These updates refine the empirical estimates that guide future selections.

4 Results

We implement **REASONING COMPILER** as an extension to MetaSchedule [8]. 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 the LLM’s 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 a hardware-informed cost model, and backpropagation for tree statistics updates.

4.1 Experimental Setup

We evaluate **REASONING COMPILER** on five representative computational kernels drawn from production-scale models: (1) self-attention layer from Llama-3-8B [16], (2) mixture-of-experts (MoE) layer from DeepSeek-R1 [17], (3) self-attention layer from FLUX (stable diffusion) [18], (4) convolution layer from FLUX [18], and (5) MLP Layer from Llama-4-Scout [19]. Compiler optimization is framed as a sequential decision process and guided by MCTS [20], using the Upper Confidence bounds applied to Trees (UCT) criterion [11] with exploration parameter $c = \sqrt{2}$ and branching factor $B = 2$, following prior work [21, 22]. During search, the LLM (OpenAI GPT-4o mini [23]) 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) TVM MetaSchedule [8], which uses Evolutionary Search; (2) MCTS without LLM guidance (MCTS); and (3) **REASONING COMPILER** that uses prompt-based proposal generation (LLM-Guided MCTS). All experiments are conducted using Apache TVM v0.20.0 [9, 24]. Our main experimental environment is a dedicated Intel Core i9 workstation under a fixed software and hardware stack to isolate scheduling effects. This main environment covers all five kernels above, and is the ablation environment. To show portability and scalability across consumer and datacenter processors, we evaluate each of the five kernels on a total of five hardware platforms: Amazon Graviton2, AMD EPYC 7R13, Apple M2 Pro, Intel Core i9, and Intel Xeon E3. We also report end-to-end Llama-3-8B results across the same five platforms. 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.

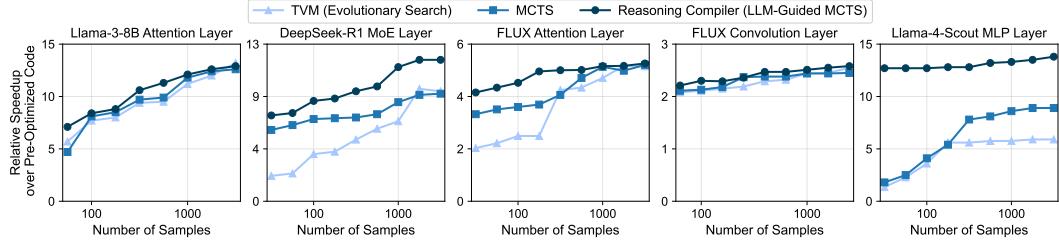


Figure 3: Relative speedup over pre-optimized code as a function of evaluated transformation proposals. **REASONING COMPILER achieves superior sample efficiency, discovering high-quality code with fewer samples across all operators in low-budget regimes.**

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 transforma-

tion proposals. Figure 3 presents results across five 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: *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 code. For instance, on the Llama-3-8B Attention Layer, LLM-Guided MCTS achieves a $7.08\times$ 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 Llama-4-Scout MLP Layer, the gap is even more pronounced: LLM-Guided MCTS achieves $12.7\times$ speedup at 20 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 Attention Layer, LLM-Guided MCTS attains a $2\times$ speedup using only 36 samples, while Evolutionary Search requires more than 600 samples, a $16\times$ reduction in tuning cost. On the FLUX Convolution Layer, LLM-Guided MCTS consistently outperforms Evolutionary Search across nearly all budget levels and reaches Evolutionary Search’s final performance after evaluating just 400 samples.

Speedup relative to baselines. LLM-Guided MCTS not only produces better code, but does so *more aggressively and earlier* in the search process. For example, on the DeepSeek-R1 MoE Layer, LLM-Guided MCTS achieves a $3.3\times$ speedup over Evolutionary Search at 36 samples; on the Llama-

Table 1: Sample efficiency comparison between REASONING COMPILER and TVM on layer-wise benchmarks across various hardware platforms.

Hardware Platform	Benchmark	TVM		Reasoning Compiler		Improvement	
		# Samples	Speedup	# Samples	Speedup	Sample Reduction	Sample Efficiency Gain
Amazon Graviton2	Llama-3-8B Attention Layer	510	$3.9\times$	60	$5.1\times$	$8.5\times$	$11.1\times$
	Deepseek-R1 MoE Layer	980	$2.7\times$	150	$5.9\times$	$6.5\times$	$14.4\times$
	FLUX Attention Layer	320	$1.6\times$	130	$3.0\times$	$2.5\times$	$4.6\times$
	FLUX Convolution Layer	160	$1.8\times$	20	$4.1\times$	$8.0\times$	$18.2\times$
	Llama-4-Scout MLP Layer	1,630	$1.7\times$	500	$4.0\times$	$3.3\times$	$7.7\times$
AMD EPYC 7R13	Llama-3-8B Attention Layer	1,400	$2.1\times$	200	$12.1\times$	$7.0\times$	$40.3\times$
	Deepseek-R1 MoE Layer	2,290	$1.7\times$	330	$2.3\times$	$6.9\times$	$9.4\times$
	FLUX Attention Layer	2,460	$1.5\times$	230	$3.1\times$	$10.7\times$	$22.1\times$
	FLUX Convolution Layer	2,520	$1.3\times$	470	$4.8\times$	$5.4\times$	$19.6\times$
	Llama-4-Scout MLP Layer	510	$6.4\times$	100	$10.2\times$	$5.1\times$	$8.1\times$
Apple M2 Pro	Llama-3-8B Attention Layer	1,010	$3.3\times$	190	$9.7\times$	$5.3\times$	$15.6\times$
	Deepseek-R1 MoE Layer	1,040	$2.8\times$	230	$4.8\times$	$4.5\times$	$7.8\times$
	FLUX Attention Layer	270	$2.1\times$	50	$3.7\times$	$5.4\times$	$9.5\times$
	FLUX Convolution Layer	2,260	$1.5\times$	510	$5.5\times$	$4.4\times$	$16.2\times$
	Llama-4-Scout MLP Layer	2,460	$2.2\times$	440	$3.4\times$	$5.6\times$	$8.6\times$
Intel Core i9	Llama-3-8B Attention Layer	920	$10.5\times$	130	$11.0\times$	$7.1\times$	$7.4\times$
	Deepseek-R1 MoE Layer	1,632	$9.1\times$	192	$9.1\times$	$8.5\times$	$8.5\times$
	FLUX Attention Layer	1,000	$5.1\times$	150	$5.4\times$	$6.7\times$	$7.0\times$
	FLUX Convolution Layer	400	$2.3\times$	72	$2.3\times$	$5.6\times$	$5.6\times$
	Llama-4-Scout MLP Layer	230	$5.6\times$	20	$12.7\times$	$11.5\times$	$26.1\times$
Intel Xeon E3	Llama-3-8B Attention Layer	2,760	$3.9\times$	320	$5.8\times$	$8.6\times$	$12.8\times$
	Deepseek-R1 MoE Layer	1,000	$3.7\times$	180	$4.4\times$	$5.6\times$	$6.6\times$
	FLUX Attention Layer	1,340	$1.4\times$	450	$3.4\times$	$3.0\times$	$7.1\times$
	FLUX Convolution Layer	220	$1.9\times$	40	$2.2\times$	$5.5\times$	$6.4\times$
	Llama-4-Scout MLP Layer	1,200	$2.0\times$	300	$6.1\times$	$4.0\times$	$12.2\times$
Geomean	—	—	$2.7\times$	—	$5.0\times$	$5.8\times$	$10.8\times$

4-Scout MLP Layer, LLM-Guided MCTS achieves a $9.3\times$ speedup over Evolutionary Search at 20 samples. 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 and MLP layer, 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, **REASONING COMPILER** consistently matches or exceeds baseline performance with fewer samples, underscoring its effectiveness across diverse operator characteristics.

Sample efficiency across hardware platforms. As shown in Table 1, **REASONING COMPILER** demonstrates superior sample efficiency compared to TVM across five hardware platforms on five benchmarks. We define sample efficiency as the speedup achieved per sample ($\frac{\text{Speedup}}{\# \text{ of Samples}}$). On average, across all 25 platform-operator pairs, **REASONING COMPILER** achieves a $5.0\times$ speedup using $5.8\times$ fewer samples, resulting in a $10.8\times$ improvement in sample efficiency. The performance gains are particularly significant for compute-intensive workloads. For instance, for the Llama-3-8B Attention Layer benchmark on AMD EPYC 7R13, **REASONING COMPILER** achieved a $12.1\times$ speedup in just 200 samples, while TVM required 1,400 samples to reach a $2.1\times$ speedup. This represents a $7.0\times$ sample reduction and a $40.3\times$ sample efficiency gain. On Intel Core i9, **REASONING COMPILER** often matches or exceeds TVM’s peak with fewer trials: on the Llama-4-Scout MLP Layer benchmark, **REASONING COMPILER** used $11.5\times$ fewer samples for a $26.1\times$ efficiency gain.

Table 2: Sample efficiency comparison between REASONING COMPILER and TVM on the end-to-end Llama-3-8B benchmark across various hardware platforms.

Hardware Platform	TVM		Reasoning Compiler		Improvement	
	# Samples	Speedup	# Samples	Speedup	Sample Reduction	Sample Efficiency Gain
Amazon Graviton2	4,560	$3.7\times$	1,440	$5.1\times$	$3.2\times$	$4.4\times$
AMD EPYC 7R13	410	$2.0\times$	140	$2.2\times$	$2.9\times$	$3.2\times$
Apple M2 Pro	4,820	$2.2\times$	1,770	$3.9\times$	$2.7\times$	$4.8\times$
Intel Core i9	3,800	$2.2\times$	720	$4.9\times$	$5.3\times$	$11.8\times$
Intel Xeon E3	4,640	$5.0\times$	670	$5.0\times$	$6.9\times$	$6.9\times$
Geomean	—	$2.8\times$	—	$4.0\times$	$3.9\times$	$5.6\times$

End-to-end sample efficiency. For the end-to-end Llama-3-8B benchmark across the five hardware platforms in Table 2, **REASONING COMPILER**’s sample efficiency improvement over TVM ranges from $3.2\times$ on AMD EPYC to $11.8\times$ on Intel Core i9. End-to-end speedups range from $2.2\times$ on AMD EPYC to $5.1\times$ on Amazon Graviton2. **REASONING COMPILER** consistently achieves significantly higher speedups: using $3.9\times$ fewer samples, it achieves a $4.0\times$ speedup and yields a $5.6\times$ geometric-mean sample efficiency improvement over TVM.

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 code with orders-of-magnitude fewer samples, our framework offers both practical deployment advantages and a compelling alternative to conventional, sample-inefficient compilation pipelines.

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 4(a) shows the relative speedup over unoptimized code as a function of the number of schedule samples evaluated by LLM-Guided MCTS on the Llama-3-8B Attention Layer benchmark using a range of LLM models

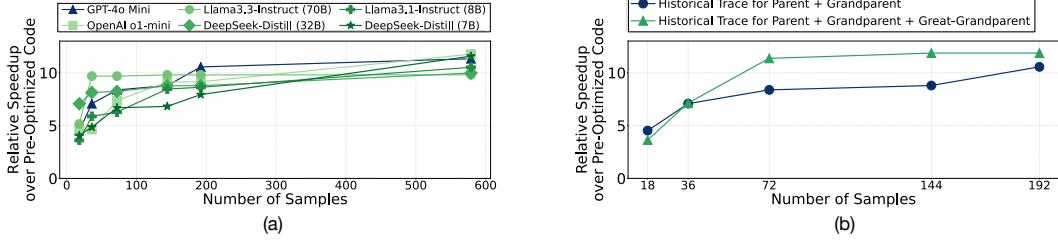


Figure 4: Ablation studies on LLM-Guided MCTS for the Llama-3-8B Attention Layer benchmark. (a) Comparing different LLMs as proposal engines shows stronger LLMs lead to faster convergence. (b) Increasing the prompt’s historical trace depth improves sample efficiency.

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. The general trend of the results supports 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 $9.69\times$ speedup after only 36 samples, roughly 86% of the GPT-4o mini’s maximum speedup but with less than 6% of its sampling budget. This corresponds to an approximately $15\times$ 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 $7.07\times$ speedup at 18 samples and reaching $9.98\times$ 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 $5.87\times$ speedup, and DeepSeek-R1-Distill-Qwen (7B) achieves a $4.86\times$ speedup at just 36 samples. When compared to the widely used Evolutionary Search strategy, which requires around 72 samples to achieve a $7.0\times$ 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 achieves 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 models in 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 4(b) presents the relative speedup over unoptimized code as a function of the number of schedule samples evaluated by LLM-Guided MCTS on the Llama-3-8B 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 $7.13\times$, slightly surpassing the $7.08\times$ of the shallower trace. However, by 72 samples, the deeper trace saturates at $11.36\times$ speedup, while the shallower trace reaches only $8.38\times$, requiring many more samples

(around 579) to approach $11.3\times$ 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

Superoptimization. While our high-level goal of discovering highly efficient program variants shares motivation with the superoptimization literature, our formulation and tractability differ substantially. Superoptimization aims to find the globally optimal instruction-level program, typically via enumerative [1, 3], symbolic [2], or stochastic [4] search over low-level assembly variants; hybrid [25] and neural [26] approaches have also been explored. STOKE [4] showed that high-quality programs often reside in low-probability regions and made the leap to use randomized search (MCMC). Neural compilers followed suit and relied on evolutionary search or simulated annealing algorithms [5–8]. In contrast, **REASONING COMPILER** treats optimization as a planning problem that leverages MCTS to reason contextually about dependencies among transformations over structured intermediate representations.

ML-Based Autotuning. Autotuning frameworks optimize performance-critical parameters (e.g., loop tile sizes, phase orderings, memory layouts) using a variety of ML-based techniques, including linear models [27, 28], tree-based methods [29, 30], Bayesian networks [31, 32], evolutionary algorithms [29, 33, 34], clustering [28, 34], and reinforcement learning [10, 33–35]. **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 [36–43] and low-level code generation [5, 9, 44–49]. Systems such as TVM/Ansor [6, 7, 9] and FlexTensor [50] 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. **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 [51–56], fuzzing [57], bug repair [58], and even high-level optimization [59]. Recent work has explored the use of LLMs to generate phase orderings or perform disassembly [60, 61]. **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

Compiling neural workloads remains a bottleneck for scalable model serving: traditional compilers struggle with combinatorial transformation spaces, and the state-of-the-art neural compilers rely on stochastic search, lacking sample efficiency and contextual awareness. This paper introduced **REASONING COMPILER**, a novel framework that formulates compiler optimization as a sequential, context-aware decision process, pairing LLM-generated proposals with MCTS and performance feedback to reason and navigate through the optimization space efficiently. By enabling LLM reasoning in the compiler optimization process, we achieve a leap from randomized search to informed and guided compilation. Our results show that **REASONING COMPILER** consistently yields faster runtimes with markedly fewer evaluations without any retraining. These gains directly translate to reduced operational cost of LLM services, lower energy usage per query, improved system responsiveness, more agile model deployment, faster model training, and accelerated innovation cycles, among other benefits. Looking ahead, the same LLM that guides compilation can accelerate its own inferencing, creating a virtuous, self-optimizing cycle in which sped-up LLMs enable more efficient transformations and progressively better models and services.

Acknowledgments

We thank the anonymous reviewers for their insightful feedback. This work was in part supported by the National Science Foundation (NSF) award CCF #2107598. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation thereon. The views contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied by the U.S. Government.

References

- [1] Henry Massalin. Superoptimizer: a look at the smallest program. In *ASPLOS*, 1987.
- [2] Rajeev Joshi, Greg Nelson, and Keith Randall. Denali: a goal-directed superoptimizer. In *PLDI*, 2002.
- [3] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In *ASPLOS*, 2006.
- [4] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *ASPLOS*, 2013.
- [5] 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.
- [6] 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.
- [7] 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.
- [8] 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.
- [9] 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.
- [10] Byung Hoon Ahn, Prannoy Pilligundla, and Hadi Esmaeilzadeh. Chameleon: Adaptive code optimization for expedited deep neural network compilation. In *ICLR*, 2020.
- [11] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *ECML*, 2006.
- [12] 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.
- [13] Minjia Zhang, Menghao Li, Chi Wang, and Mingqin Li. Dynature: Dynamic tensor program optimization in deep neural network compilation. In *ICLR*, 2021.
- [14] Byung Hoon Ahn, Sean Kinzer, and Hadi Esmaeilzadeh. Glimpse: Mathematical embedding of hardware specification for neural compilation. In *DAC*, 2022.
- [15] Perry Gibson and José Cano. Transfer-tuning: Reusing auto-schedules for efficient tensor program code generation. In *PACT*, 2023.
- [16] 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.
- [17] 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.
- [18] Black Forest Labs. Flux. <https://github.com/black-forest-labs/flux>, 2024.
- [19] Meta. The Llama 4 herd: The beginning of a new era of natively multimodal AI innovation. <https://ai.meta.com/blog/llama-4-multimodal-intelligence/>, 2025.
- [20] 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.

- [21] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *CG*, 2007.
- [22] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2–3):235–256, 2002.
- [23] OpenAI. Openai GPT-4o mini API. <https://platform.openai.com/docs/models/gpt-4o-mini>, 2025.
- [24] Apache TVM Community. Apache TVM v0.20.0. <https://github.com/apache/tvm/releases/tag/v0.20.0>, 2025.
- [25] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. Scaling up superoptimization. In *ASPLOS*, 2016.
- [26] Rudy Bunel, Alban Desmaison, M. Pawan Kumar, Philip H. S. Torr, and Pushmeet Kohli. Learning to superoptimize programs. In *ICLR*, 2017.
- [27] Mark Stephenson and Saman Amarasinghe. Predicting unroll factors using supervised classification. In *CGO*, 2005.
- [28] 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.
- [29] Douglas Simon, John Cavazos, Christian Wimmer, and Sameer Kulkarni. Automatic construction of inlining heuristics using machine learning. In *CGO*, 2013.
- [30] 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.
- [31] 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.
- [32] 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.
- [33] Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and David Li. Mlgo: a machine learning guided compiler optimizations framework. *arXiv*, 2021.
- [34] Haolin Pan, Yuanyu Wei, Mingjie Xing, Yanjun Wu, and Chen Zhao. Towards efficient compiler auto-tuning: Leveraging synergistic search spaces. In *CGO*, 2025.
- [35] 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.
- [36] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. Deep learning with dynamic computation graphs. *arXiv*, 2017.
- [37] 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.
- [38] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. Optimizing cnn model inference on cpus. In *ATC*, 2019.
- [39] 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.
- [40] Yaoyao Ding, Ligeng Zhu, Zhihao Jia, Gennady Pekhimenko, and Song Han. Ios: Inter-operator scheduler for cnn acceleration. In *MLSys*, 2021.
- [41] 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.
- [42] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. Equality saturation for tensor graph superoptimization. In *MLSys*, 2021.
- [43] 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.
- [44] 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.

- [45] Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodik, and Vinod Grover. Fireiron: A data-movement-aware scheduling language for gpus. In *PACT*, 2020.
- [46] Jian Weng, Animesh Jain, Jie Wang, Leyuan Wang, Yida Wang, and Tony Nowatzki. Unit: Unifying tensorized instruction compilation. In *CGO*, 2021.
- [47] 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.
- [48] Wookeun Jung, Thanh Tuan Dao, and Jaejin Lee. Deepcuts: A deep learning optimization framework for versatile gpu workloads. In *PLDI*, 2021.
- [49] 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.
- [50] 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.
- [51] 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.
- [52] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Noumane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. Starcoder 2 and the stack v2: The next generation. *arXiv*, 2024.
- [53] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Empowering code generation with oss-instruct. *arXiv*, 2023.
- [54] 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.
- [55] 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.
- [56] 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.
- [57] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Universal fuzzing via large language models. *arXiv*, 2023.
- [58] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *ICSE*, 2023.
- [59] Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob R. Gardner, Yiming Yang, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning performance-improving code edits. In *ICLR*, 2024.
- [60] 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.
- [61] 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 a 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: $vj = \dots + j_1 * 64 + j_2 * 64$

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: $vj = \dots + j_1 * 256 + j_2 * 64$

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 your reasoning and your suggested transformations.

For example, your answer should be in the following format:

Reasoning: This code still has large loop extents, so I'd tile it twice differently, then unroll.

Transformations to apply: TileSize, TileSize, Unroll.

LLM Response:

Reasoning: The current schedule tiles the j-axis as $2048=4\times 8\times 1\times 64$, whereas the parent uses $4\times 2\times 4\times 64$. The higher predicted score of the current node (0.773 vs. 0.313) suggests that expanding outer parallel granularity ($j_1=8$) helped, but collapsing j_2 to 1 removed a small register-level tile, limiting unrolling and hurting reuse within each $j_3=64$ chunk.

I would (1) reintroduce a non-unit j_2 while shrinking j_1 so that $j_1\times j_2 \approx 8$; for instance, [4, 4, 2, 64] keeps enough outer tasks but creates a $j_2\times j_3 = 128$ -wide microtile that unrolls cleanly.
(2) Tile the 7168-step k-reduction to align with $j_3=64$, e.g., 7168 = $7\times 16\times 64$; the 16-step inner slice can be unrolled to cut loop overhead and improve register reuse across the j_3 tile.
(3) Move the compute of block "moe" to the j_1 level so each j_1 tile initializes/accumulates C locally, shortening live ranges and reducing write-back pressure.
(4) Parallelize the outer j_1 (or t_1 if j_1 becomes too small) to utilize cores without touching the reduction.
(5) Unroll the small j_2 loop and the k-inner microtile. These changes retain the beneficial outer structure of the current node while restoring a compact inner tile for ILP and cache locality, which should further improve throughput.

Transformations to apply: TileSize, TileSize, ComputeLocation, Parallel, Unroll, Unroll.

B Sample Efficiency and Speedup Results

Table 3 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. This table corresponds to Figure 3 in the paper.

Table 3: 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
Llama-3-8B 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	8.50	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	
DeepSeek-R1 MoE Layer	Number of Samples	36	54	72	144	192	600	900	1632	3000	
	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	
FLUX Attention Layer	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	
	Evolutionary Search	2.22	2.44	2.73	2.73	4.64	4.71	5.11	5.61	5.58	
FLUX Convolution Layer	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	72	150	200	400	600	1000	1600	3000	
Llama-4-Scout MLP Layer	Evolutionary Search	2.08	2.11	2.15	2.19	2.29	2.32	2.44	2.45	2.55	
	MCTS	2.11	2.13	2.18	2.37	2.38	2.38	2.44	2.44	2.45	
	MCTS + LLM	2.21	2.30	2.29	2.36	2.47	2.47	2.51	2.55	2.58	
	Number of Samples	20	50	100	250	400	600	1000	1500	3000	
	Evolutionary Search	1.36	2.28	3.61	5.59	5.59	5.75	5.76	5.94	5.94	
	MCTS	1.76	2.51	4.05	5.41	7.83	8.13	8.58	8.90	8.90	
	MCTS + LLM	12.74	12.74	12.74	12.75	12.75	13.24	13.26	13.52	13.79	

C Impact of LLM Choice and Reasoning Strategy

As a continuation of Figure 4(a), Table 4 reports speedup over unoptimized code on three additional benchmarks: DeepSeek-R1 MoE Layer, FLUX 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 4: 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 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 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 Layer	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 4(b), Table 5 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 $9.39\times$ speedup at just 18 samples compared to $6.14\times$ for the shallower context. Similarly, on Llama-3-8B Attention Layer, the extended context leads to a higher final speedup ($11.87\times$ vs. $11.33\times$) and earlier convergence. The performance gains, while smaller, are also consistent on FLUX 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 5: Speedup over unoptimized code across varying numbers of samples for different context lengths.

		Number of Samples	18	36	72	150	200	600
Llama-3-8B Attention Layer	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	
	Number of Samples	18	36	72	150	200	600	
DeepSeek-R1 MoE Layer	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	
	Number of Samples	18	36	72	150	200	600	
FLUX Attention Layer	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	
	Number of Samples	18	36	72	150	200	600	
FLUX Convolution Layer	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	
	Number of Samples	18	36	72	150	200	600	

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 6, 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 [21, 22]. 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 6: Speedup over unoptimized code across varying numbers of samples for different branching factors.

	Number of Samples	18	36	72	150	200	600
Llama-3-8B 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
DeepSeek-R1 MoE Layer	Number of Samples	18	36	72	150	200	600
	B = 2	6.14	7.05	8.33	8.53	9.10	9.45
FLUX Attention Layer	Number of Samples	18	36	72	150	200	600
	B = 2	4.09	4.48	4.89	5.37	5.42	5.43
FLUX Convolution Layer	Number of Samples	18	36	72	150	200	600
	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

F Cost of LLMs Used in Experiments

In Table 7, for each benchmark, we report the API cost of running a full experiment with every LLM used to generate transformation proposals. We run a high number of samples to understand the boundary of performance improvements and allow the algorithm to saturate. For OpenAI, our main results used GPT-4o mini, the lowest-cost model available at submission time. For open-source models, we used Hugging Face APIs through the Nscale hyperscaler provider. Across benchmarks, these open-source models achieved competitive speedups and sample efficiency relative to GPT-4o mini, indicating that open-source models are a viable alternative when commercial APIs are impractical. Costs of open-source models could be further reduced by local deployment.

Table 7: Cost of different LLM APIs per entire experiment (USD) across layer-wise and end-to-end benchmarks.

Layer / Task	Model					
	GPT-4o mini	OpenAI o1-mini	Llama3.3- Instruct (70B)	DeepSeek- Distill (32B)	Llama3.1- Instruct (8B)	DeepSeek- Distill (7B)
Llama-3-8B Attention Layer	\$0.89	\$6.56	\$2.07	\$1.55	\$0.31	\$2.07
DeepSeek-R1 MoE Layer	\$0.90	\$6.63	\$2.09	\$1.57	\$0.31	\$2.09
FLUX Attention Layer	\$0.88	\$6.47	\$2.03	\$1.52	\$0.30	\$2.03
FLUX Convolution Layer	\$1.12	\$8.25	\$2.67	\$2.00	\$0.40	\$2.67
Llama-4-Scout MLP Layer	\$0.90	—	—	—	—	—
End-to-End Llama-3-8B	\$1.59	—	—	—	—	—

G LLM Proposal Validity and Fallback Rates

LLM-generated transformations can occasionally be syntactically valid but semantically redundant or performance-regressive. During any single MCTS expansion, proposals that fail basic validity checks (e.g., naming or use-context non-compliance) are simply discarded while the remaining valid proposals proceed, and no fallback is triggered. A fallback occurs only when all LLM-generated proposals in that expansion are invalid, in which case the search reverts to the default, non-LLM expansion policy and continues without interruption. In Table 8, we report the fallback rate as the average fraction of expansions that trigger this non-LLM path (i.e., expansions in which all LLM proposals are invalid). To prevent downstream harm from poor but valid transformations, the cost model evaluates all proposed transformations before they are added to the tree; proposals with low estimated values are naturally pruned. Because the transformation space is a known, finite set of legal rewrites, most correctness issues reduce to naming compliance and use-context, which modern instruction-tuned LLMs typically handle well. Empirically, commercial models (GPT-4o mini and OpenAI o1-mini) show 0% fallback rates, larger open-source models perform similarly (Llama3.3-Instruct 70B at 0.08% and DeepSeek-Distill 32B at 0.17%), whereas smaller models exhibit higher fallback rates (Llama3.1-Instruct 8B at 10.50% and DeepSeek-Distill 7B at 17.20%).

Table 8: Fallback rate by model used as the transformation proposal generator.

Model	Fallback Rate
GPT-4o mini	0%
OpenAI o1-mini	0%
Llama3.3-Instruct (70B)	0.08%
DeepSeek-Distill (32B)	0.17%
Llama3.1-Instruct (8B)	10.50%
DeepSeek-Distill (7B)	17.20%