Improving Assembly Code Performance with Large Language Models via Reinforcement Learning

Anonymous Author(s)
Affiliation
Address
`email`

Abstract

Large language models (LLMs) have demonstrated strong performance across a wide range of programming tasks, yet their potential for code optimization remains underexplored. This work investigates whether LLMs can optimize the performance of assembly code, where fine-grained control over execution enables improvements that are difficult to express in high-level languages. We present a reinforcement learning framework that trains LLMs using Proximal Policy Optimization (PPO), guided by a reward function that considers both functional correctness, validated through test cases, and execution performance relative to the industry-standard compiler `gcc -O3`. To support this study, we introduce a benchmark of 8,072 real-world programs. Our model, `Qwen2.5-Coder-7B-PPO`, achieves 96.0% test pass rates and an average speedup of 1.47× over the `gcc -O3` baseline, outperforming all 20 other models evaluated, including `Claude-3.7-sonnet`. These results indicate that reinforcement learning can unlock the potential of LLMs to serve as effective optimizers for assembly code performance.

1 Introduction

Recent advances in large language models (LLMs) have achieved state-of-the-art solutions across a wide range of programming tasks [1–5]. However, their potential for program optimization remains underexplored. Generating highly optimized code is critical in performance-sensitive domains, and prior work has investigated the use of LLMs to optimize C++ and Python programs [6–8]. In this work, we aim to utilize LLMs to improve the performance of assembly code, extending their capabilities beyond optimization for high-level languages.

Assembly code optimization is traditionally the responsibility of compilers. While modern compilers apply a series of rule-based transformations to improve performance, such a design introduces the classic phase ordering problem [9], where the order of optimizations can substantially affect the performance of the generated code. Due to the inherent complexity of the optimization task, especially the vast space of possible transformation sequences, compilers face fundamental challenges in converging to optimal code, often leaving significant performance on the table [10].

An alternative approach is superoptimization, which searches the space of all programs that are functionally equivalent to the compiler's output, aiming to identify the most performant variant. In principle, this strategy may yield optimal code. However, the search space grows exponentially with program size, making exhaustive exploration computationally infeasible in practice. Furthermore, prior work on superoptimization [11, 12] has primarily targeted loop-free, straight-line code, where it is more tractable to formally verify that the optimized code is semantically equivalent to the original. As a result, these approaches are not directly applicable to most real-world programs with loops.

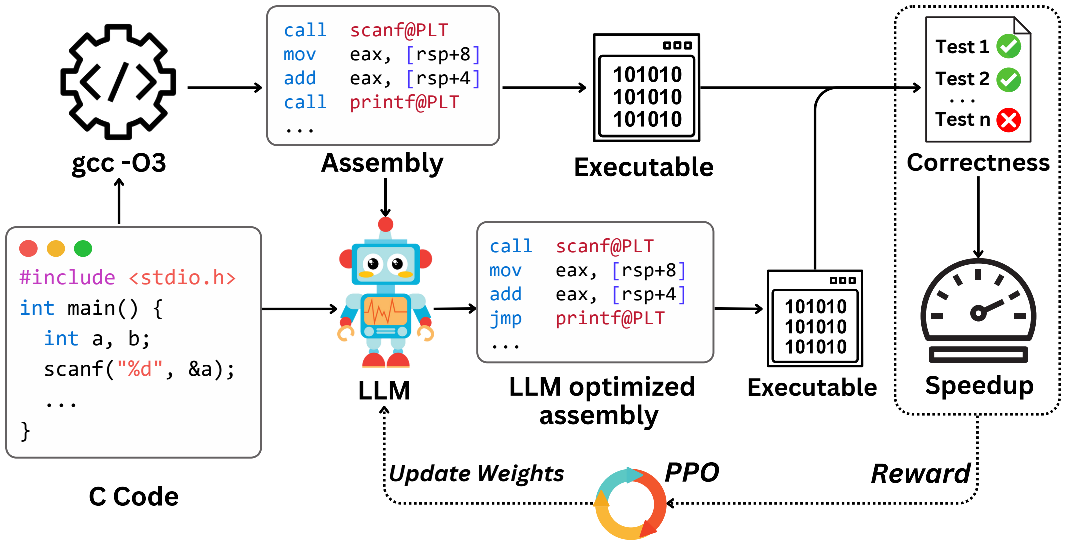

Figure 1: Overview of the assembly code optimization task. Given a C program and its baseline assembly from `gcc -O3`, an LLM is fine-tuned with Proximal Policy Optimization (PPO) to generate improved assembly. The reward function reflects correctness and performance based on test execution.

In this work, we explore using large language models to optimize the performance of assembly code. Compared with high-level languages such as Python or C++, assembly code operates closer to the hardware, offering fine-grained control over execution and enabling optimizations that are difficult to express or realize in higher-level code. However, this setting poses several challenges. Assembly code is relatively rare and may be underrepresented in pretraining corpora [13], making it harder for LLMs to reason effectively about their behavior. Furthermore, industry compilers such as GCC have been extensively tuned by performance engineers over decades. Achieving additional speedups beyond `gcc -O3` (the compiler's highest optimization level) is a technically challenging task.

To address the challenges of low-level code optimization, we apply reinforcement learning to enhance the ability of LLMs to optimize assembly code. As shown in Figure 1, we use Proximal Policy Optimization (PPO) to train an LLM using a reward function that considers both correctness and performance. Correctness is evaluated based on whether the generated code passes program-specific test cases, and performance is measured by its speedup relative to the baseline produced by `gcc -O3`. To support this setting, we construct a new dataset of 8,072 assembly programs derived from real-world competitive programming submissions. Each instance includes input-output test cases and baseline assembly code generated by the compiler at the highest optimization level, which serves as the starting point for further optimization.

We evaluate our approach on the proposed benchmark and find that reinforcement learning substantially improves the ability of LLMs to optimize assembly code. Starting from the base model `Qwen2.5-Coder-7B-Instruct`, which achieves a modest 1.10x speedup over the `gcc -O3` baseline, our PPO-trained model reaches 1.47× average speedup and improves both compile and test pass rates to 96.0%. It achieves the strongest performance across all evaluation metrics, outperforming all 20 other models evaluated, including `Claude-3.7-sonnet`. Ablation studies show that reward functions emphasizing final speedup, rather than intermediate correctness signals, lead to more effective training.

In summary, our contributions are as follows:

- We introduce the task of optimizing assembly code performance using large language models, aiming for fine-grained performance improvements beyond what traditional compiler optimizations can achieve.
- We construct a dataset of real-world C programs paired with the corresponding assembly code generated by the `gcc -O3` baseline. Using this dataset, we explore improving LLMs on this task through reinforcement learning, applying Proximal Policy Optimization (PPO).

- We evaluate 21 LLMs on the proposed benchmark and show that our training substantially improves performance: `Qwen2.5-Coder-7B-PPO` achieves the highest compile and test pass rates, as well as the best average speedup (1.47×) over the `gcc -O3` baseline, outperforming all other models (including `Claude-3.7-sonnet`) across all evaluation metrics.

2 Related Work

Large Language Models for Code. Benchmarks for evaluating large language models (LLMs) on code generation from natural language specifications have received increasing attention. Notable examples include HumanEval [1], MBPP [2], APPS [3], and more recent efforts [14–17]. In parallel, many models have been developed to enhance code generation capabilities, such as Codex [1], AlphaCode [18], CodeGen [19], InCoder [20], StarCoder [21], DeepSeek-Coder [22], Code Llama [23], and others [24, 25]. Beyond code generation, LLMs have been applied to real-world software engineering tasks including automated program repair [26, 27], software testing [28, 29], bug localization [30], and transpilation [31, 32]. SWE-bench [4] integrates these tasks into a benchmark for resolving real GitHub issues. Building on this, SWE-agent [5] and subsequent works [33, 34] employ an agent-based framework that leverages LLMs to improve the issue resolution process.

Recent work has also explored LLMs for improving program performance. CodeRosetta [35] targets automatic parallelization, such as translating C++ to CUDA. Other efforts focus on optimizing Python code for efficiency [7, 8] or enabling self-adaptation [36], and improving C++ performance [6]. Of particular relevance are approaches to low-level code optimization [37, 38]. The LLM Compiler foundation models [39, 40] are primarily designed for code size reduction and binary disassembly, whereas our work focuses on optimizing assembly code for performance. LLM-Vectorizer [41] offers a formally verified solution for auto-vectorization, a specific compiler pass. In contrast, our work does not restrict the optimization type and uses test-case validation.

Learning-Based Code Optimization. The space of code optimization is vast, and many approaches have leveraged machine learning to improve program performance. A classic challenge in compilers is the phase-ordering problem, where performance depends heavily on the sequence of optimization passes. AutoPhase [42] uses deep reinforcement learning to tackle this, while Coreset [43] employs graph neural networks (GNNs) to guide optimization decisions. Modern compilers apply extensive rewrite rules but offer no guarantee of optimality. Superoptimization seeks the most efficient program among all semantically equivalent variants of the compiler output. Traditional methods use stochastic search, such as Markov Chain Monte Carlo [11], with follow-up work improving scalability [44, 12] and extending to broader domains [45, 46]. These rely on formal verification for correctness, restricting them to small, loop-free programs. In contrast, our approach uses test-based validation, enabling optimization of general programs with loops. With the rise of deep learning, substantial attention has turned to optimizing GPU kernel code. AutoTVM [47] pioneered statistical cost model-based search for CUDA code optimization, followed by methods such as Ansor [48], AMOS [49], and other recent systems [50–52].

More recently, using LLMs as code optimizers has gained popularity [6, 53, 37], with growing attention to reinforcement learning approaches that guide LLMs through reward-based feedback [54, 34]. CodeRL [55] incorporates unit test-based rewards within an actor-critic framework [56], while PPOCoder [57] extends this with Proximal Policy Optimization (PPO) [58], along with other variants [59]. Subsequent efforts have adapted RL-based techniques [60] to additional low-resource programming languages, including Verilog [61]. To the best of our knowledge, our work is the first to apply reinforcement learning to optimize assembly code using LLMs. Assembly code offers fine-grained control and potential for significant performance gains, but it remains underexplored due to limited training data and the complexity of low-level semantics.

3 Methodology

3.1 Task Definition

Let C be a program written in a high-level language such as C. A modern compiler like `gcc` can compile C into an x86-64 assembly program $P = \texttt{gcc}(C)$, which can then be further assembled into an executable binary. The assembly program P serves as an intermediate representation that exposes low-level optimization opportunities, making it suitable for aggressive performance improvement. We assume the semantics-preserving nature of the compilation process, i.e., $[\![C]\!] = [\![P]\!]$, so that the behavior of the assembly program P is identical to that of the source program C.

In theory, the goal is to produce a program P' that is functionally equivalent to P across the entire input space $\mathcal{X}$, i.e., $P(x) = P'(x)$ for all $x \in \mathcal{X}$. Since verifying this property is undecidable in general, we approximate equivalence using a finite test set $\mathcal{T} = \{(x_i, y_i)\}_{i=1}^{n}$, where each input-output pair (x_i, y_i) captures the expected behavior of C.

We say that an assembly program P' is *valid* if it can be successfully assembled and linked into an executable binary. Let $\texttt{valid}(P') \in \{\texttt{True}, \texttt{False}\}$ denote this property. We define the set of *correct* programs as:

$$\mathcal{S}(P) = \{P' \mid \texttt{valid}(P') \ \wedge \ \forall (x_i, y_i) \in \mathcal{T},\ P'(x_i) = y_i\}\,.$$

Performance and Speedup. Let $t(P)$ denote the execution time of P on the test set $\mathcal{T}$, and let $t(P')$ be the corresponding execution time for P'. The speedup of P' relative to P is defined as:

$$\text{Speedup}(P') = \begin{cases} \frac{t(P)}{t(P')} & \text{if } P' \in \mathcal{S}(P) \text{ and } t(P') < t(P), \\ 1 & \text{otherwise.} \end{cases}$$

Optimization Objective. The objective is to generate a candidate program P' that maximizes Speedup(P'). Only programs in $\mathcal{S}(P)$ are eligible for speedup; any candidate that fails to compile into a binary or produces incorrect outputs is assigned a default speedup of 1. This reflects a practical fallback: when the generated program is invalid, the system can revert to the baseline P, compiled with `gcc -O3`, which defines the 1× reference performance. Although $\mathcal{S}(P)$ captures the correctness criteria, we do not restrict the LLM to generate only valid programs. Instead, the model produces arbitrary assembly code, and correctness is verified post hoc via compilation and test execution. We train an LLM using reinforcement learning (see Section 3.3) to generate candidates that both satisfy correctness and achieve performance improvements.

3.2 Dataset Construction

We construct our dataset using C programs from CodeNet [62], a large-scale corpus of competitive programming submissions. Each dataset instance is a tuple $(C, P, \mathcal{T})$, where C is the original C source code, $P = \texttt{gcc\_O3}(C)$ is the corresponding x86-64 assembly generated by compiling C with `gcc` at the `-O3` optimization level, and $\mathcal{T} = \{(x_i, y_i)\}_{i=1}^{n}$ is the test set. Since not all CodeNet problems include test inputs, we adopt those provided by prior work [18] to define x_i, but discard their output labels. Instead, we regenerate each output y_i by executing the original submission on input x_i, as many CodeNet programs are not accepted solutions, and even accepted ones do not reliably pass all test cases.

Given the scale of CodeNet, which contains over 8 million C and C++ submissions, we sample a subset for this study. To focus on performance-critical cases, we sample programs that exhibit the highest relative speedup from `gcc -O0` (no optimization) to `gcc -O3` (maximum optimization). Such strategy serves two purposes: (1) it favors programs with complex logic that lead to suboptimal performance under `-O0` and can be effectively optimized by `-O3`, and (2) it creates a more challenging setting by starting from code that has already benefited from aggressive compiler optimizations. If an LLM can generate code that further improves upon `gcc -O3`, it suggests that the model can outperform the compiler's "expert" solution. The final dataset consists of 7,872 training programs and 200 held-out evaluation programs, with additional statistics provided in Section 4.

3.3 Reinforcement Learning

We conceptualize our task as a standard contextual multi-armed bandit problem [63], defined by a context space $\mathcal{S}$, an action space $\mathcal{A}$, and a reward function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. Each context $s \in \mathcal{S}$ represents a problem instance, comprising the source program C, its baseline assembly P, and the associated test cases $\mathcal{T}$. An action $a \in \mathcal{A}$ corresponds to generating a candidate assembly program $\tilde{P}$. The reward function $r(s, a)$ evaluates the quality of the generated program based on correctness and performance. We will describe different designs of the reward function later. A policy $\pi : \mathcal{S} \to \Delta(\mathcal{A})$

maps a context s to a probability distribution over actions and samples an action $a \in \mathcal{A}$ stochastically. Given a distribution μ over problem instances, the expected performance of a policy π under reward function r is expressed as $\mathbb{E}_{s\sim\mu, a\sim\pi(\cdot|s)}[r(s,a)]$. The objective is to find a policy that maximizes this expected reward.

Optimization with PPO. We train the policy using *Proximal Policy Optimization* (PPO) [58], a first-order policy-gradient algorithm that stabilizes training by constraining each policy update to remain close to the previous one. Specifically, PPO maximizes a clipped surrogate objective of the form $\mathbb{E}_{s,a}\left[\min\left(\rho(\theta)\hat{A},\ \mathrm{clip}(\rho(\theta), 1-\epsilon, 1+\epsilon)\,\hat{A}\right)\right]$, where $\rho(\theta) = \pi_\theta(a \mid s)/\pi_{\theta_{\mathrm{old}}}(a \mid s)$ is the probability ratio between the current and previous policy, $\hat{A}$ is the estimated advantage of action a in state s, and ϵ is a clipping coefficient that limits the policy update to a small trust region. We use a critic model to estimate $\hat{A}$, and compute rewards based on the correctness and execution time of the generated program, eliminating the need for a separate reward model.

Reward Function Design. As defined in our contextual bandit setup, the reward function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ assigns a scalar score to each (context, action) pair. Each context $s \in \mathcal{S}$ consists of the source program C, the baseline assembly P, and a test set $\mathcal{T} = \{(x_i, y_i)\}_{i=1}^{n}$. An action $a \in \mathcal{A}$ corresponds to a generation procedure that produces a candidate assembly program $\tilde{P} = \mathtt{gen}(a)$.

We define two auxiliary metrics for computing reward:

$$\mathrm{pass}(s,a) = \tfrac{1}{|\mathcal{T}|}\sum_{(x,y)\in\mathcal{T}} \mathbf{1}[\tilde{P}(x) = y], \quad \mathrm{speedup}(s,a) = t(P)/t(\tilde{P}),$$

which respectively denote the fraction of test cases passed and the speedup of the generated program $\tilde{P}$ relative to the baseline P. We evaluate two reward function variants:

1. **Correctness-Guided Speedup (CGS):**

$$r(s,a) = \begin{cases} -1, & \text{if } \tilde{P} \text{ fails to compile,} \\ \mathrm{pass}(s,a), & \text{if some tests fail,} \\ 1 + \alpha \cdot \mathrm{speedup}(s,a), & \text{if all tests pass.} \end{cases}$$

2. **Speedup-Only (SO):**

$$r(s,a) = \begin{cases} 0, & \text{if } \tilde{P} \text{ fails to compile or any test fails,} \\ \mathrm{speedup}(s,a), & \text{otherwise.} \end{cases}$$

In CGS, the constant α controls the relative importance of speedup once full correctness is achieved (i.e., all test cases pass). The CGS reward provides a dense signal by assigning intermediate credit for successful compilation and partially correct outputs, guiding the policy even when the final objective is not yet met. In contrast, SO defines a more direct and sparse objective, assigning nonzero reward only to programs that are both correct and performant, thereby rewarding only the terminal goal of achieving speedup.

4 Experimental Setup

Dataset. We describe our dataset construction approach in Section 3.2. Each instance consists of a C source program C, the corresponding `gcc -O3` compiled assembly P, and a set of test cases $\mathcal{T}$ for correctness evaluation. The final dataset contains 7,872 training programs and 200 evaluation programs, with average program lengths and test case counts summarized in Table 1.

Split	# Prog.	Avg. Tests	Avg. LOC	
			C	Assembly
Training	7,872	8.86	22.3	130.3
Evaluation	200	8.92	21.9	133.3

Table 1: Dataset statistics across training and evaluation splits. LOC = lines of code.

Prompts. For each instance, we construct a prompt that includes the original C program along with the generated assembly using `gcc -O3`. All test cases are withheld from the model. The model is instructed to generate only the optimized x86-64 assembly code. We show the prompt template in Appendix A.2.

Metrics. We evaluate each model using both correctness and performance metrics. *Compile pass* is the percentage of problems for which the generated assembly compiles to binary executable successfully, while *test pass* is the percentage of problems where the compiled code passes all test cases. For a given problem, any single failed test case is considered a failure for the test pass metric. Both metrics are computed across the entire validation set. For performance, we measure the relative speedup over the `gcc -O3` baseline. As defined in Section 3.1, we assign a default speedup of 1× to any candidate that fails to compile, fails any test case, or is slower than the baseline. This reflects the practical setting where a system can fall back to the `gcc -O3` output, resulting in no performance gain. We report the *25th*, *50th* (median), and *75th* percentiles of speedup to capture distributional behavior, along with the *average speedup* over the entire evaluation set.

Models. We evaluate 21 state-of-the-art language models spanning a diverse range of architectures. Our benchmark includes frontier proprietary models such as `gpt-4o` [64], `o4-mini`, `gemini-2.0-flash-001` [65], and `claude-3.7-sonnet`, as well as open-source families such as Llama [66], DeepSeek [67], and Qwen [25]. In addition, we include models distilled from DeepSeek-R1 [68] based on Qwen and Llama. Finally, we evaluate recent compiler foundation models [39, 40] that are pre-trained on assembly code, building upon Code Llama and designed specifically for compiler-related tasks. All open-source models are instruction-tuned.

Performance Measurement. To ensure an accurate performance evaluation, we use `hyperfine` [69], a benchmarking tool that reduces measurement noise by performing warmup runs followed by repeated timed executions. For each program's execution, we discard the first three runs and report the average runtime over the next ten runs.

Implementation. We implement our customized reinforcement learning reward functions within the VERL framework [70], which enables fine-tuning of LLMs using Proximal Policy Optimization (PPO). As part of this setup, we build a task-specific environment that handles program compilation, test execution, and runtime measurement, as detailed in Section 3.3. This environment provides the model with direct scalar feedback based on both functional correctness and execution performance.

Training Configurations. Among all evaluated models (see Table 2), we select `Qwen2.5-Coder-7B-Instruct` for training due to its strongest correctness results and substantial room for performance improvement, while intentionally avoiding compiler-specific foundation models to preserve generality. Training is performed on a single node with four A100 GPUs. Full hyperparameter settings are provided in Appendix A.1.

5 Results

5.1 Evaluation of Different Models

Table 2 presents results across evaluated models. Most models struggle to generate performant assembly: the majority yield only 1.00× speedup, with low compile and test pass rates. Among baseline models, `claude-3.7-sonnet` and `DeepSeek-V3` perform best, achieving test pass rates above 40% and average speedups of 1.22× and 1.21×, respectively. Notably, some models such as `DeepSeek-R1` fail to generate any valid assembly, and `o4-mini` achieves only 4.5% test pass. These results underscore the difficulty of the task and motivate the need for a task-specific approach.

Compiler foundation models (prefixed with `llm-compiler-`) are pretrained on assembly code and compiler intermediate representations. Among them, `llm-compiler-13b` demonstrates strong performance in both correctness and speedup. In contrast, the fine-tuned variants (`-ftd`) perform poorly, likely because they are adapted for tasks such as disassembling x86-64 and ARM assembly into LLVM-IR, rather than optimizing assembly code for execution performance.

Model	Compile Pass	Test Pass	Speedup Percentiles 25th	50th	75th	Average Speedup
DS-R1-Distill-Qwen-1.5B	0.0%	0.0%	1.00×	1.00×	1.00×	1.00×
DeepSeek-R1	0.0%	0.0%	1.00×	1.00×	1.00×	1.00×
DS-R1-Distill-Llama-70B	5.5%	0.0%	1.00×	1.00×	1.00×	1.00×
DS-R1-Distill-Qwen-14B	11.5%	0.5%	1.00×	1.00×	1.00×	1.00×
gpt-4o-mini	44.5%	1.0%	1.00×	1.00×	1.00×	1.00×
Llama-4-Maverick-17B	77.5%	7.0%	1.00×	1.00×	1.00×	1.02×
Llama-3.2-11B	84.0%	21.0%	1.00×	1.00×	1.00×	1.02×
gpt-4o	81.0%	5.0%	1.00×	1.00×	1.00×	1.02×
Llama-4-Scout-17B	68.5%	5.5%	1.00×	1.00×	1.00×	1.02×
o4-mini	25.0%	4.5%	1.00×	1.00×	1.00×	1.02×
gemini-2.0-flash-001	57.5%	4.0%	1.00×	1.00×	1.00×	1.03×
Qwen2.5-72B	59.5%	7.5%	1.00×	1.00×	1.00×	1.03×
Llama-3.2-90B	82.5%	15.0%	1.00×	1.00×	1.00×	1.05×
Qwen2.5-Coder-7B	79.0%	61.0%	1.00×	1.00×	1.00×	1.10×
DeepSeek-V3	94.0%	43.0%	1.00×	1.00×	1.40×	1.21×
claude-3.7-sonnet	94.5%	58.5%	1.00×	1.10×	1.45×	1.22×
llm-compiler-7b-ftd	2.0%	2.0%	1.00×	1.00×	1.00×	1.00×
llm-compiler-13b-ftd	2.5%	2.0%	1.00×	1.00×	1.00×	1.01×
llm-compiler-7b	55.0%	54.0%	1.00×	1.00×	1.00×	1.09×
llm-compiler-13b	60.5%	59.5%	1.00×	1.27×	1.63×	1.34×
Qwen2.5-Coder-7B-PPO (Ours)	**96.0%**	**96.0%**	**1.21×**	**1.42×**	**1.66×**	**1.47×**

Table 2: Comparison of LLMs on our assembly optimization benchmark. We report compilation success rate, test pass rate, and average speedup over the `gcc -O3` baseline. All open-source models are instruction-tuned. We evaluate general-purpose foundational models, compiler-specific foundation models, and our PPO-trained model, which improves average speedup from 1.10× to 1.47×.

We select `Qwen2.5-Coder-7B-Instruct` for RL training due to its strong compile pass rate (79.0%) and highest test pass rate (61.0%) among models. After PPO fine-tuning, it achieves 96.0% on both metrics and increases average speedup from 1.10× to 1.47×. Notably, it is the only model to exhibit meaningful speedup even at the 25th percentile, and it outperforms all other models across all evaluation metrics, including correctness, average speedup, and speedup percentiles.

5.2 Ablation Study of Reward Function Design

We evaluate two reward designs for RL: *Correctness-Guided Speedup* (CGS) and *Speedup-Only* (SO). CGS penalizes compilation failures, rewards partial correctness, and scales final reward by speedup once all tests pass. SO uses speedup as the sole reward, but only when all tests pass.

Method	Compile Pass	Test Pass	Avg. Speedup
Base Model	79.0%	61.0%	1.10x
RL w/ CGS	95.5%	94.5%	1.38×
RL w/ SO	96.0%	96.0%	1.47×

Table 3: Ablation study comparing reward function variants. CGS provides intermediate reward shaping, while SO uses a sparse and terminal signal.

As shown in Table 3, both variants achieve high compile pass rates and test pass rates, but SO yields better performance. Removing intermediate shaping appears to help the model focus on terminal objectives. We also tried varying the CGS scaling factor α (5 or 10) and found that it has a negligible effect.

These results indicate that sparse, terminal rewards (SO) are more effective in this setting. Since the base model already reaches 61.0% test pass, correctness is not the bottleneck; optimizing directly for speedup offers a stronger training signal.

Method	Compile Pass		Test Pass		Avg. Speedup	
	w/ O3	w/o O3	w/ O3	w/o O3	w/ O3	w/o O3
DeepSeek-V3	94.0%	25.5%	43.0%	4.5%	1.21×	1.02×
claude-3.7-sonnet	94.5%	53.0%	58.5%	16.0%	1.22×	1.07×
llm-compiler-7b	55.0%	12.0%	54.0%	0.0%	1.09×	1.00×
llm-comiler-13b	60.5%	2.0%	59.5%	0.5%	1.34×	1.00×
Qwen2.5-Coder-7B	79.0%	0.0%	61.0%	0.0%	1.10×	1.00×
Qwen2.5-Coder-7B-PPO	96.0%	0.0%	96.0%	0.0%	1.47×	1.00×

Table 4: Ablation study on the impact of including `gcc -O3` baseline assembly in the prompt. Each metric is reported with and without access to the baseline assembly generated by the compiler.

5.3 Can LLMs Directly Compile Programs without Baseline Assembly?

In our main evaluation, we always include the baseline assembly generated by `gcc -O3` in the prompt. While this baseline offers a strong starting point for further optimization, it may also bias the model toward replicating patterns from the compiler's output. In this subsection, we investigate a more challenging setup: *Can large language models directly compile C code into performant assembly without relying on the compiler-generated baseline?*

We compare two settings in evaluation: (1) providing both the C source and the `gcc -O3` assembly (default, "w/ O3" in Table 4), and (2) providing only the C source ("w/o O3"). For each model, we report compile success rate, test pass rate, and average speedup.

Table 4 shows that removing the baseline assembly leads to severe degradation. For instance, `Qwen2.5-Coder-7B-PPO` drops from 96.0% correctness and 1.47× speedup to 0.0% and 1.00×, respectively. Even strong models like `Claude-3.7-sonnet` suffer substantial declines.

These results suggest that direct compilation from C to optimized assembly remains challenging for current LLMs. The compiler output provides a reliable reference for LLMs. This supports our framework design: using `gcc -O3` as an effective starting point for reinforcement learning. While future work may explore direct generation from C, we expect it to be substantially more challenging due to the absence of compiler guidance.

5.4 Case Study

Figure 2 presents a representative example where large language models (LLMs), including gpt-4o and claude-3.7-sonnet, discover an optimization that outperforms a state-of-the-art compiler. The original C function computes the population count (i.e., the number of set bits) by repeatedly shifting the input and accumulating its least significant bit. The assembly code produced by `gcc -O3` preserves this loop structure, relying on explicit bitwise operations and conditional branches to compute the result.

In contrast, the LLM generates a significantly more concise and efficient implementation that replaces the entire loop with a single `popcnt` instruction. This instruction, supported by modern x86-64 architectures, performs the same computation in one operation, thereby reducing both instruction count and runtime overhead.

Such a transformation is beyond the reach of `gcc -O3`, which applies a predetermined sequence of rule-based optimization passes and does not conduct semantic-level rewrites of this kind. In this case, the language model is able to synthesize functionally equivalent code that exploits hardware-level instructions not utilized by the compiler. This demonstrates the potential of language models to optimize assembly by exploring a broader space of semantics-preserving program transformations.

6 Discussion

Limitations. A key limitation of our approach is the absence of formal correctness guarantees. Although we validate generated programs using input-output test cases, such testing is inherently incomplete and may overlook edge cases. Consequently, unlike fully verified systems such as Stoke [11], LLM-generated assembly may produce wrong results or exhibit undefined behavior. This

C Code

```c
int f(unsigned long x)
{
  int res = 0;
  while (x > 0)
  {
    res += x & 1;
    x >>= 1;
  }
  return res;
}
```

GCC -O3 Output

```
.L0:
 xorl %eax, %eax
 testq %rdi, %rdi
 je .L2
.L1:
 movq %rdi, %rdx
 andl $0x1, %edx
 addq %rdx, %rax
 shrq $0x1, %rdi
 jne .L1
 retq
.L2:
 retq
```

LLM Generated

```
.L0:
 popcnt %rdi, %rax
 retq
```

Figure 2: Case study comparing the C code, baseline assembly produced by `gcc -O3`, and optimized assembly generated by the LLM. The model successfully replaces the loop with the specialized hardware instruction `popcnt`, resulting in a significantly more concise implementation.

limitation reflects a broader challenge in programming languages: verifying the semantic equivalence between two arbitrary programs is undecidable in the general case.

Another limitation lies in the inherent randomness of performance measurement on real hardware. Although we mitigate noise through repeated measurements, low-level hardware fluctuations can still introduce variability into speedup estimates. Such nondeterminism is difficult to eliminate entirely in real-world settings. While prior work has adopted simulator-based evaluation [6], simulators may fail to faithfully capture the actual hardware performance.

Finally, the observed performance gains may not generalize across machines. The model may implicitly learn and exploit hardware-specific characteristics like cache size. As a result, a model trained on one machine may not retain its effectiveness when deployed on a different machine.

Future Work. While we use Proximal Policy Optimization (PPO), future work may explore alternative reinforcement learning algorithms such as GRPO [71]. Expanding to larger and more diverse datasets, particularly those involving performance-critical code beyond competitive programming, would make the setting more realistic and applicable. Combining reinforcement learning with supervised fine-tuning may also be beneficial, although it remains unclear whether training on `gcc -O3` outputs would provide additional gains. Another direction is to incorporate an interactive refinement loop, where the model iteratively updates its output using feedback from errors or performance measurements. Finally, extending our approach from x86-64 to other architectures such as MIPS, ARM, or GPU programming could broaden its applicability and impact.

7 Conclusion

We explore the use of large language models (LLMs) for optimizing assembly code, a setting where fine-grained control over execution enables performance improvements that are difficult to express in high-level languages. While traditional compilers rely on fixed rule-based transformations, they face fundamental limitations due to the complexity of the optimization space. To address this, we apply reinforcement learning to fine-tune LLMs with Proximal Policy Optimization (PPO), using a reward function based on correctness and speedup over the `gcc -O3` baseline. To support this effort, we introduce a benchmark of 8,072 real-world C programs with compiler-generated baseline assembly and test cases. Our resulting model, `Qwen2.5-Coder-7B-PPO`, achieves the highest compile and test pass rates (96.0%) and the best average speedup (1.47×), outperforming all 20 other models evaluated across all metrics. These results indicate that reinforcement learning can unlock the potential of LLMs to serve as effective optimizers for assembly code performance.

References

[1] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, “Evaluating large language models trained on code,” *arXiv preprint arXiv:2107.03374*, 2021.

[2] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, “Program synthesis with large language models,” *arXiv preprint arXiv:2108.07732*, 2021.

[3] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song *et al.*, “Measuring coding challenge competence with apps,” *arXiv preprint arXiv:2105.09938*, 2021.

[4] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, “Swe-bench: Can language models resolve real-world github issues?” *arXiv preprint arXiv:2310.06770*, 2023.

[5] J. Yang, C. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, “Swe-agent: Agent-computer interfaces enable automated software engineering,” *Advances in Neural Information Processing Systems*, vol. 37, pp. 50 528–50 652, 2024.

[6] A. Shypula, A. Madaan, Y. Zeng, U. Alon, J. Gardner, M. Hashemi, G. Neubig, P. Ranganathan, O. Bastani, and A. Yazdanbakhsh, “Learning performance-improving code edits,” *arXiv preprint arXiv:2302.07867*, 2023.

[7] M. Du, A. T. Luu, B. Ji, Q. Liu, and S.-K. Ng, “Mercury: A code efficiency benchmark for code large language models,” *arXiv preprint arXiv:2402.07844*, 2024.

[8] J. Liu, S. Xie, J. Wang, Y. Wei, Y. Ding, and L. Zhang, “Evaluating language models for efficient code generation,” *arXiv preprint arXiv:2408.06450*, 2024.

[9] T. J. W. I. R. Center, F. Allen, and J. Cocke, *A Catalogue of Optimizing Transformations*. IBM Thomas J. Watson Research Center, 1971. [Online]. Available: https://books.google.com/books?id=oeXaZwEACAAJ

[10] T. Theodoridis, M. Rigger, and Z. Su, “Finding missed optimizations through the lens of dead code elimination,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 697–709.

[11] E. Schkufza, R. Sharma, and A. Aiken, “Stochastic superoptimization,” *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 305–316, 2013.

[12] R. Bunel, A. Desmaison, M. P. Kumar, P. H. Torr, and P. Kohli, “Learning to superoptimize programs,” *arXiv preprint arXiv:1611.01787*, 2016.

[13] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei *et al.*, “Starcoder 2 and the stack v2: The next generation,” *arXiv preprint arXiv:2402.19173*, 2024.

[14] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, “Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation,” in *Advances in Neural Information Processing Systems*, 2023.

[15] J. Li, G. Li, X. Zhang, Y. Zhao, Y. Dong, Z. Jin, B. Li, F. Huang, and Y. Li, “Evocodebench: An evolving code generation benchmark with domain-specific evaluations,” *Advances in Neural Information Processing Systems*, vol. 37, pp. 57 619–57 641, 2024.

[16] C. S. Xia, Y. Deng, and L. Zhang, “Top leaderboard ranking= top coding proficiency, always? evoeval: Evolving coding benchmarks via llm,” *arXiv preprint arXiv:2403.19114*, 2024.

[17] T. Y. Zhuo, M. C. Vu, J. Chim, H. Hu, W. Yu, R. Widyasari, I. N. B. Yusuf, H. Zhan, J. He, I. Paul *et al.*, “Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions,” *arXiv preprint arXiv:2406.15877*, 2024.

[18] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[19] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

[20] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," *arXiv preprint arXiv:2204.05999*, 2022.

[21] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.

[22] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[23] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[24] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang, "Magicoder: Empowering code generation with oss-instruct," *arXiv preprint arXiv:2312.02120*, 2023.

[25] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu *et al.*, "Qwen2. 5-coder technical report," *arXiv preprint arXiv:2409.12186*, 2024.

[26] C. S. Xia and L. Zhang, "Less training, more repairing please: revisiting automated program repair via zero-shot learning," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 959–971.

[27] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1482–1494.

[28] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Universal fuzzing via large language models," *CoRR*, 2023.

[29] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, "Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries," in *Proceedings of the 46th IEEE/ACM international conference on software engineering*, 2024, pp. 1–13.

[30] A. Z. Yang, C. Le Goues, R. Martins, and V. Hellendoorn, "Large language models for test-free fault localization," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–12.

[31] Z. Yang, F. Liu, Z. Yu, J. W. Keung, J. Li, S. Liu, Y. Hong, X. Ma, Z. Jin, and G. Li, "Exploring and unleashing the power of large language models in automated code translation," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1585–1608, 2024.

[32] S. Bhatia, J. Qiu, N. Hasabnis, S. Seshia, and A. Cheung, "Verified code transpilation with llms," *Advances in Neural Information Processing Systems*, vol. 37, pp. 41 394–41 424, 2024.

[33] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, "Agentless: Demystifying llm-based software engineering agents," *arXiv preprint arXiv:2407.01489*, 2024.

[34] Y. Wei, O. Duchenne, J. Copet, Q. Carbonneaux, L. Zhang, D. Fried, G. Synnaeve, R. Singh, and S. I. Wang, "Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution," *arXiv preprint arXiv:2502.18449*, 2025.

[35] A. TehraniJamsaz, A. Bhattacharjee, L. Chen, N. K. Ahmed, A. Yazdanbakhsh, and A. Jannesari, "Coderosetta: Pushing the boundaries of unsupervised code translation for parallel programming," *arXiv preprint arXiv:2410.20527*, 2024.

[36] D. Huang, J. Dai, H. Weng, P. Wu, Y. Qing, H. Cui, Z. Guo, and J. Zhang, "Effilearner: Enhancing efficiency of generated code via self-optimization," *Advances in Neural Information Processing Systems*, vol. 37, pp. 84 482–84 522, 2024.

[37] A. Wei, A. Nie, T. S. Teixeira, R. Yadav, W. Lee, K. Wang, and A. Aiken, "Improving parallel program performance through dsl-driven code generation with llm optimizers," *arXiv preprint arXiv:2410.15625*, 2024.

[38] A. Ouyang, S. Guo, S. Arora, A. L. Zhang, W. Hu, C. Ré, and A. Mirhoseini, "Kernelbench: Can llms write efficient gpu kernels?" *arXiv preprint arXiv:2502.10517*, 2025.

[39] C. Cummins, V. Seeker, D. Grubisic, B. Roziere, J. Gehring, G. Synnaeve, and H. Leather, "Meta large language model compiler: Foundation models of compiler optimization," *arXiv preprint arXiv:2407.02524*, 2024.

[40] ——, "Llm compiler: Foundation language models for compiler optimization," in *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction*, 2025, pp. 141–153.

[41] J. Taneja, A. Laird, C. Yan, M. Musuvathi, and S. K. Lahiri, "Llm-vectorizer: Llm-based verified loop vectorizer," in *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, 2025, pp. 137–149.

[42] A. Haj-Ali, Q. J. Huang, J. Xiang, W. Moses, K. Asanovic, J. Wawrzynek, and I. Stoica, "Autophase: Juggling hls phase orderings in random forests with deep reinforcement learning," *Proceedings of machine learning and systems*, vol. 2, pp. 70–81, 2020.

[43] Y. Liang, K. Stone, A. Shameli, C. Cummins, M. Elhoushi, J. Guo, B. Steiner, X. Yang, P. Xie, H. J. Leather *et al.*, "Learning compiler pass orders using coreset and normalized value prediction," in *International Conference on Machine Learning*. PMLR, 2023, pp. 20 746–20 762.

[44] P. M. Phothilimthana, A. Thakur, R. Bodik, and D. Dhurjati, "Scaling up superoptimization," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016, pp. 297–310.

[45] R. Sharma, E. Schkufza, B. Churchill, and A. Aiken, "Conditionally correct superoptimization," *ACM SIGPLAN Notices*, vol. 50, no. 10, pp. 147–162, 2015.

[46] B. Churchill, R. Sharma, J. Bastien, and A. Aiken, "Sound loop superoptimization for google native client," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 313–326, 2017.

[47] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to optimize tensor programs," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[48] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen *et al.*, "Ansor: Generating {High-Performance} tensor programs for deep learning," in *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, 2020, pp. 863–879.

[49] S. Zheng, R. Chen, A. Wei, Y. Jin, Q. Han, L. Lu, B. Wu, X. Li, S. Yan, and Y. Liang, "Amos: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 874–887.

[50] J. Shao, X. Zhou, S. Feng, B. Hou, R. Lai, H. Jin, W. Lin, M. Masuda, C. H. Yu, and T. Chen, "Tensor program optimization with probabilistic programs," *Advances in Neural Information Processing Systems*, vol. 35, pp. 35 783–35 796, 2022.

[51] Y. Zhao, H. Sharif, V. Adve, and S. Misailovic, "Felix: Optimizing tensor programs with gradient descent," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2024, pp. 367–381.

[52] M. Wu, X. Cheng, S. Liu, C. Shi, J. Ji, K. Ao, P. Velliengiri, X. Miao, O. Padon, and Z. Jia, "Mirage: A multi-level superoptimizer for tensor programs," *arXiv preprint arXiv:2405.05751*, 2024.

[53] D. Grubisic, C. Cummins, V. Seeker, and H. Leather, "Compiler generated feedback for large language models," *arXiv preprint arXiv:2403.14714*, 2024.

[54] S. Dou, Y. Liu, H. Jia, L. Xiong, E. Zhou, W. Shen, J. Shan, C. Huang, X. Wang, X. Fan *et al.*, "Stepcoder: Improve code generation with reinforcement learning from compiler feedback," *arXiv preprint arXiv:2402.01391*, 2024.

[55] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. H. Hoi, "Coderl: Mastering code generation through pretrained models and deep reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 35, pp. 21 314–21 328, 2022.

[56] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," *Advances in neural information processing systems*, vol. 12, 1999.

[57] P. Shojaee, A. Jain, S. Tipirneni, and C. K. Reddy, "Execution-based code generation using deep reinforcement learning," *arXiv preprint arXiv:2301.13816*, 2023.

[58] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[59] J. Liu, Y. Zhu, K. Xiao, Q. Fu, X. Han, W. Yang, and D. Ye, "Rltf: Reinforcement learning from unit test feedback," *arXiv preprint arXiv:2307.04349*, 2023.

[60] J. Liu, T. Nguyen, M. Shang, H. Ding, X. Li, Y. Yu, V. Kumar, and Z. Wang, "Learning code preference via synthetic evolution," *arXiv preprint arXiv:2410.03837*, 2024.

[61] N. Wang, B. Yao, J. Zhou, X. Wang, Z. Jiang, and N. Guan, "Large language model for verilog generation with code-structure-guided reinforcement learning," 2025. [Online]. Available: https://arxiv.org/abs/2407.18271

[62] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker *et al.*, "Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks," *arXiv preprint arXiv:2105.12655*, 2021.

[63] T. Lu, D. Pál, and M. Pál, "Contextual multi-armed bandits," in *Proceedings of the Thirteenth international conference on Artificial Intelligence and Statistics*. JMLR Workshop and Conference Proceedings, 2010, pp. 485–492.

[64] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[65] G. Team, R. Anil, S. Borgeaud, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth, K. Millican *et al.*, "Gemini: a family of highly capable multimodal models," *arXiv preprint arXiv:2312.11805*, 2023.

[66] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[67] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan *et al.*, "Deepseek-v3 technical report," *arXiv preprint arXiv:2412.19437*, 2024.

[68] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi *et al.*, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," *arXiv preprint arXiv:2501.12948*, 2025.

[69] "Hyperfine," 2025. [Online]. Available: https://github.com/sharkdp/hyperfine

[70] G. Sheng, C. Zhang, Z. Ye, X. Wu, W. Zhang, R. Zhang, Y. Peng, H. Lin, and C. Wu, "Hybridflow: A flexible and efficient rlhf framework," *arXiv preprint arXiv:2409.19256*, 2024.

[71] Z. Shao, P. Wang, Q. Zhu, R. Xu, J. Song, X. Bi, H. Zhang, M. Zhang, Y. Li, Y. Wu *et al.*, "Deepseekmath: Pushing the limits of mathematical reasoning in open language models," *arXiv preprint arXiv:2402.03300*, 2024.

A Appendix

A.1 Training Configurations

Component	Setting
Base model	`Qwen2.5-Coder-7B-Instruct`
Actor's learning rate	1e-6
Critic's learning rate	1e-5
Batch size	16
Epoch	1
Max prompt length	2000 tokens
Max response length	2000 tokens
Gradient checkpointing	Enabled (both actor and critic)
Rollout temperature	0.5
Hardware	4× A100 GPUs

Table A1: Key training configurations for VERL PPO fine-tuning.

A.2 Prompt Template

Prompt Template

```
Given the following C code and assembly code, your task is to generate highly
optimized x86-64 assembly code.

C Code:

<C code here>

Assembly Code:

<baseline assembly code here produced by gcc -O3>

Only output the optimized assembly code.  Do not include any other text.
Do not write any comments in the assembly code.  Wrap the assembly code in
assembly tags.
Optimized Assembly Code:
```

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: We have verified that the claims accurately reflect our contributions and the defined research scope.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We discuss the limitations in Section 6.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

Justification: The paper does not include theoretical results.

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: We clearly discuss the experiment setup in Section 4 with additional details in Appendix A.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [No]

Justification: The code and data are being prepared for release and will be made publicly available with full instructions upon acceptance.

Guidelines:

- The answer NA means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyper-parameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: We provide the experimental details in Section 4 and Appendix A.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: We run each performance measurement 10 times with 3 warmup runs, and report 25th, 50th, 75th percentiles and average speedup to capture variability.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).

- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: We provide such details in Section 4 and Appendix A.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: We have reviewed the NeurIPS Code of Ethics and confirm that our research complies fully with its principles.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [No]

    Justification: The paper focuses on techniques for code optimization. We do not explicitly discuss societal impacts, as no direct applications or harms are identified.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.

- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: The paper does not release any models or datasets with high risk of misuse.

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: We use publicly available datasets intended for research use (e.g., CodeNet), and all sources of data and models are properly cited with adherence to their license terms.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.

- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [No]

    Justification: We plan to release the dataset and model with documentation upon acceptance, but they are not publicly available at submission time.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: The paper does not involve crowdsourcing or research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: The paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

Answer: [Yes]

Justification: The paper investigates using LLMs to optimize assembly code performance, making LLM usage central to the core methodology.

Guidelines:

- The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.