Agentic Code Optimization via Compiler-LLM Cooperation

BENJAMIN MIKEK, AWS AI, USA and Georgia Institute of Technology, USA
DANYLO VASHCHILENKO*, AWS AI, USA
BRYAN LU, AWS AI, USA
PANPAN XU, AWS AI, USA

Generating performant executables from high level languages is critical to software performance across a wide range of domains. Modern compilers perform this task by passing code through a series of well-studied optimizations at progressively lower levels of abstraction, but may miss optimization opportunities that require high-level reasoning about a program's purpose. Recent work has proposed using LLMs to fill this gap. While LLMs can achieve large speedups on some programs, they frequently generate code that is incorrect. In this work, we propose a method to balance the correctness of conventional compiler optimizations with the "creativity" of LLM-based code generation: compiler-LLM cooperation. Our approach integrates existing compiler optimization passes with LLM-based code generation at multiple levels of abstraction, retaining the best features of both types of code optimization. We realize our approach with a multi-agent system that includes (1) LLM-based optimization agents for each level of abstraction, (2) individual compiler constituents as tools, (3) an LLM-based test generation agent that probes the correctness and performance of generated code, and (4) a guiding LLM that orchestrates the other components. The strategy enables LLM-based optimization of input programs at multiple levels of abstraction and introduces a method for distributing computational budget between levels. Our extensive evaluation shows that compiler-LLM cooperation outperforms both existing compiler optimizations and level-specific LLM-based baselines, producing speedups up to 1.25×.

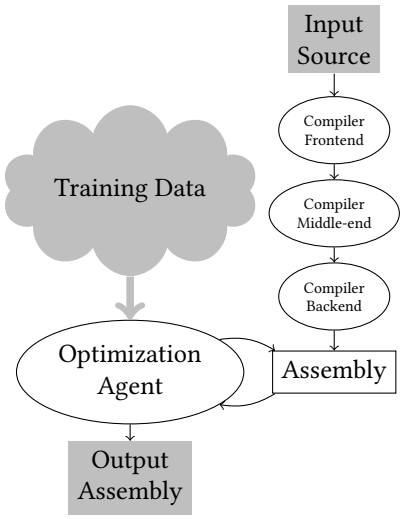
1 Introduction

Given an algorithm in a high-level language like C or PyTorch, the role of a compiler is to generate low-level executable code which is correct and efficient. Modern compilers [24] achieve both of these goals by analyzing and applying optimizations at several *levels of abstraction*; for example, by processing from source code, through an intermediate representation (IR), and finally an assembly language like x86. Some compiler components move code from one abstraction level to the next (*lowering*), while optimization passes improve a program without changing its level (*rewriting*). Compilers excel at performing optimizations that are structured, deterministic, and general, like peephole optimizations, dead code elimination, and vectorization. Decades of research on verification [26, 29] and testing [25] have made modern compilers robust and driven down rates of miscompilation to negligible levels for most applications. But since compiled programs may be run billions of times, even small improvements over existing compilers can have large payoffs.

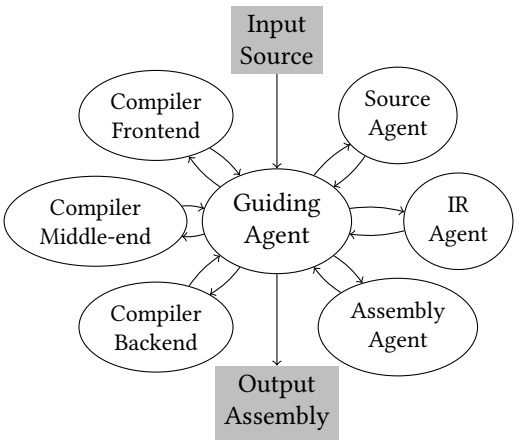
Recently, attention has turned to harnessing the power of large language models (LLMs) to augment this optimization process [15, 34, 40]. A typical workflow for LLM-based optimization [49] is shown in Figure 1a. An input program is first compiled to a particular level of abstraction, x86 assembly in this case. An LLM is then provided a context window containing the code and an optimization prompt; it produces optimized output code. LLM-generate code is often improved by iterative refinement, in which the LLM is provided feedback on the generated code and tries to improve on its own prior generation. The code generation model itself may be improved when more data is available via reinforcement learning, supervised fine-tuning, etc. LLM-based code optimization has shown promise for improving program performance at assembly level [49], at source level [15, 40], and in the GPU context [4, 31, 34].

*Corresponding author

Authors' Contact Information: Benjamin Mikek, AWS AI, New York, NY, USA and Georgia Institute of Technology, Atlanta, GA, USA, bmikek@gatech.edu; Danylo Vashchilenko, AWS AI, New York, NY, USA, vdanylo@amazon.com; Bryan Lu, AWS AI, Santa Clara, CA, USA, yuzhelu@amazon.com; Panpan Xu, AWS AI, Santa Clara, CA, USA, xupanpan@amazon.com.



(a) Existing approaches to LLM-based code optimization, focusing on one level of abstraction.



(b) Our approach, using a guiding agent, a testing agent, and level-specific optimization agents at each level of abstraction.

Fig. 1. Comparison between existing work (left) and our multi-level approach (right).

The downside of LLM-based code optimization is that models frequently generate incorrect code—rates of incorrect generation range from 10% to as high as 90% depending on model and prompting [40]. The problem is exacerbated at lower levels of abstraction, as LLMs struggle to comply with the precise syntax and correctness demands of low-level languages like assembly. Wei et al. [49], for example, report the best un-augmented model produces incorrect code 42% of the time when rewriting, and almost always fails when generating assembly code from source. Worse performance at lower levels of abstraction occurs, in part, because model training datasets focus on high-level languages: assembly code makes up just 0.08% of the popular code dataset The Stack, while C and C++ programs make up 13% [21]. Compiler IRs do not appear at all. Existing work combats low rates of correctness by utilizing in-context learning (ICL) with retrieved samples [4] and training methods like reinforcement learning (RL) [16, 49] and supervised fine-tuning (SFT) [10, 48]. These approaches, however, rely on the availability of curated prompts or training data—they trade better correctness for the availability of data. There is therefore a trade-off between LLMs and compilers. LLMs are more "creative" and can reason about more complex optimization opportunities than compilers in some instances, but often produce incorrect code—boosting correctness requires training or sampling data. Compilers on the other hand are more conservative, producing code that is correct every time but sometimes missing high-level optimization opportunities.

In this work, we propose a method to obtain the best of both worlds: compiler-LLM cooperation. Our approach balances the optimization promise of LLMs with the conservative but correct approach of compilers by allowing the two to cooperate. The key enabling insight is that LLM-based optimization can be carried out not just at a single level of abstraction, but interleaved with calls to an existing compiler across all levels of abstraction. We then introduce a novel method of choosing how best to interleave LLM-based code generation with compiler components. Our approach, shown in Figure 1b, is to construct a multi-agent system that includes core LLM agents that optimize at each level of abstraction and a guiding agent that chooses when to use a compiler and when to use LLM-based optimization. Our realization of compiler-LLM cooperation does not rely on the availability of training data, but each level-specific agent is compatible with parallel sampling [6], feedback loops [4], and other training- and test-time methods of tuning.

Conceptually, our approach has three main advantages. First, giving a multi-agent system the freedom to choose how to interleave existing compiler calls and LLM-based optimizations balances LLMs' optimization strengths with the guaranteed correctness of compilers. Second, it enables LLM-based optimization at multiple levels of abstraction, allowing a single program to benefit from multiple classes of optimization and potentially unlocking synergies among optimizations at multiple levels of abstraction. Third, the compiler-LLM cooperation strategy is useful for cases in which large training sets are not available—while our approach is *compatible* with strategies like SFT and RL, it can still produce correct code without access to data beyond the target program.

We realize compiler-LLM cooperation as ACCLAIM (**A**gentic **C**ooperation between **C**ompiler and **L**LM for **A**utomated **IM**provement of programs), a multi-agent system that optimizes C programs targeting CPUs. Our architecture, shown in Figure 1, consists of the following:

- For each level of abstraction, a level-specific agent that performs LLM-based rewriting.
- A testing agent that generates test cases and provides correctness and performance feedback.
- A guiding agent with access to each level-specific agent, the testing agent, and the lowering and rewriting components of an existing compiler as tools.

Given an input program, the guiding agent is empowered to use reasoning tokens to determine what types of optimizations might be a good match. It can call the existing compiler components, level-specific agents, and the testing agent to optimize an input program, and use feedback to inform better optimization decisions on the fly. The output is a low-level program that, in the ideal case, performs better than an assembly program generated by a conventional compiler alone.

We evaluate ACCLAIM on a standard set of C programs and find that it achieves mean speedups of 1.25× against `clang -O3`, outperforming both level-specific and naive multi-level baselines with equal computation budgets. Our analysis of ACCLAIM's behavior shows that it makes use of all compiler components and core agents and frequently backtracks or re-calls agents when results are not satisfactory. In summary, this paper makes the following primary contributions:

- We introduce a framework for LLM-based optimization across multiple levels of abstraction in a compiler pipeline.
- We develop an agentic workflow for dynamically choosing when and at which levels of abstraction to apply LLM-based optimization with an orchestration agent.
- We show that, combined, the multi-level approach and orchestration strategy outperform both single-level and naive multi-level baselines.

In the rest of this paper, Section 2 gives an example to motivate multi-level LLM-based optimization. Section 3 describes our approach in detail, and Section 4 describes our realization of compiler-LLM cooperation and evaluates its effectiveness. Section 5 discusses the limitations and implications of our results, while Section 6 surveys related work and Section 7 concludes.

2 Motivating Example

This section motivates the power of compiler-LLM cooperation in optimizing code with an example function. Throughout this paper, we provide examples using LLVM's `clang` compiler.

Input Function. Figure 2a shows the source code of a function f that that loops through values in range $[1, k]$. For each value, f counts the number of bits that are set, and then sums the results; in short, f returns the sum of "population counts" over a given range. f is initially $O(k \log k)$.

Source Level LLM Optimization. LLMs show potential for optimizing code on their own. Figure 2b shows a version of f optimized by Claude 3.7 Sonnet. Here, the LLM has introduced a single mathematical function that performs the bit counting for each value in the outer loop, leaving only the inner loop. f is now $O(\log k)$.

```
int total = 0;
for (int i = 1; i <= k; i++) {
    int x = i;
    int res = 0;
    while (x > 0) {
        res += x & 1;
        x >>= 1;
    }
    total += res;
}
```

(a) Original code on input k, containing two nested loops. The result is stored in `total`.

```
while (remaining > 0) {
    total += (k / (bit_pos * 2)) * bit_pos;
    unsigned long remainder = k % (bit_pos * 2);
    if (remainder >= bit_pos) {
        total += remainder - bit_pos + 1;
    }
    bit_pos <<= 1;
    if (bit_pos > k) break;
    remaining >>= 1;
}
```

(b) For loop after LLM-based optimization at source level.

```
%18 = load i32, ptr %5, align 4
%19 = and i32 %18, 1
%20 = load i32, ptr %6, align 4
%21 = add nsw i32 %20, %19
store i32 %21, ptr %6, align 4
%22 = load i32, ptr %5, align 4
%23 = ashr i32 %22, 1
store i32 %23, ptr %5, align 4
```

(c) LLVM IR program implementing the inner loop from f.

```
%5 = phi i64 [ 1, %3 ], [ %11, %10 ]
%6 = phi i32 [ 0, %3 ], [ %9, %10 ]
%7 = trunc i64 %5 to i32
%8 = tail call i32 @llvm.ctpop.i32(i32 %7)
%9 = add i32 %6, %8
```

(d) LLVM IR program after LLM-based optimization.

```
call <4 x i32> @llvm.ctpop.v4i32(<4 x i32> %vec.ind)
call <4 x i32> @llvm.ctpop.v4i32(<4 x i32> %step.add)
```

(e) LLVM IR program from Figure 2d after application of LLVM's existing vectorization pass.

Fig. 2. Motivating Example

LLM-based Optimization. The potential of LLM-based code optimization is not limited to the source code level. Figure 2c shows the LLVM IR code for the inner loop of function f, with two basic blocks; the rest of the program is omitted for brevity. An LLM is also capable of modifying the program at IR level, as shown in Figure 2d. Here, Claude 3.7 Sonnet is used again and identifies that the operation performed in the inner loop is bit counting. It therefore produces IR code that calls LLVM's bit-counting intrinsic `@llvm.ctpop.i32`, eliminating the inner loop and achieving $O(k)$ asymptotic runtime.[1]

[1] $O(k)$ runtime is achieved only if the underlying hardware supports bit counting in constant time; for instance, the x86-64 ISA provides POPCNT for 16-, 32-, and 64-bit integers.

Compiler-LLM Cooperation. Figure 2e shows the results of applying existing compiler optimizations to the LLM-generated code in Figure 2d; LLVM is able to vectorize the calls to the population count intrinsic, placing 8 repetitions (2 instructions with 4 operations each) of the call within each iteration of the outer loop. For a processor with the right hardware, this further reduces the running time by a factor of 4×.

The existing compiler does not introduce the `ctpop` intrinsic on its own; only the LLM does so. On the other hand, the compiler is able to correctly vectorize the calls to the intrinsic while the LLM does not take advantage of this optimization opportunity. In this way, Figure 2e shows the power of compiler-LLM cooperation: the LLM first spots an optimization that the compiler does not perform, and the compiler can subsequently perform a further optimization (vectorization) for which an LLM is ill-suited.

3 Approach

This section formally states the problem of compiler-LLM cooperation, and describes how we design it solution, including an LLM-based guiding agent, three level-specific optimization agents, and a testing agent.

3.1 Preliminaries

The problem of compiler-LLM cooperation is one of coordinating the lowering and rewriting tools provided by both compilers and LLMs. We here provide a formal statement of the problem, and in the rest of this section, describe our approach to realize compiler-LLM cooperation.

Assume we have a fixed ordered set of languages $\mathbb{L} = \{L_1, L_2, \ldots, L_n\}$, where the ordering represents levels of abstraction; for example, we may have L_1 as C source code, L_2 as compiler IR, etc. A program P written in language L_i might be transformed in one of two ways.

Definition 3.1 (Rewrite). For some $i \leq n$, a function $f : L_i \longrightarrow L_i$ is a *rewrite* at level i.

Definition 3.2 (Lowering). For some $i, j \leq n$ such that $i < j$, a function $f : L_i \longrightarrow L_j$ is a *lowering* from level i to level j.

Rewrites and lowerings are the atomic operations that move between and within levels of abstraction in a compilation pipeline. A compiler then consists of a set of rewrites and lowerings:

Definition 3.3 (Compiler). A compiler is a pair (F, S) with a finite set of functions $F = \{f_1, f_2, \ldots, f_k\}$ and a finite sequence $S \subset \mathcal{P}(F)$ such that

- For all $f \in F$, f is either a rewrite or a lowering,
- The first function of S, S_1 has domain L_1,
- The last function of S, S_f, has range L_n, and
- For each sequential pair S_k, S_{k+1} in S, if the range of S_k is a language L_i, then the domain of S_{k+1} is also L_i.

The properties of the sequence S ensure that the compiler has a path involving lowerings and rewrites that stretches from the highest level of abstraction to the lowest. In practice, S typically involves passing through every level of abstraction (no skipped levels). The existing LLVM compiler pipeline uses three levels of abstraction: C source code, LLVM IR, and x86 assembly. This compiler has $|F| = 3$:

- The frontend is a lowering from C source code to LLVM IR;
- The "middle-end" is a rewriting at LLVM IR level, including all IR optimization passes; and
- The backend is a lowering from LLVM IR to assembly, which also includes some backend optimizations.

LLVM and other compilers also perform some light source code pre-processing (resolving `#define` directives, for instance); we treat these as a constituent part of the frontend.

With the compiler in hand, we introduce level-specific optimization agents.

Definition 3.4 (Level-Specific Agent). A level-specific agent is any algorithm that implements a rewrite function $f : L_i \to L_i$.

In practice, we implement each level-specific agent with several LLM calls; Section 3.4 gives the details and variants of our level-specific agents. Last, we introduce testing agents:

Definition 3.5 (Testing Agent). For a given level of abstraction i, a testing agent is an algorithm that implements a function $T_i : L_i \to \mathbb{R} \times \mathbb{R}$ such that $T_i(p)$ produces a pair $\left(T_{\text{correct}}, T_{\text{perf}}\right)$, where $T_{\text{correct}} = 1$ if the program is correct and less than 1 otherwise, and T_{perf} represents the performance of program p.

In this work, we treat correctness as a proportion of tests passed and performance as a ratio of original runtime to final runtime. The problem of compiler-LLM cooperation is a problem of orchestrating rewrites provided by both LLMs and the compiler, lowerings provided by the compiler, and calls to the testing agent in order to achieve a correct program with good performance.

Problem Statement

Given a compiler (F, S), a set of level-specific agents $A = \{a_1, a_2, \ldots a_i\}$, a testing agent T, and an input program $p \in L_1$, construct a composition of the functions of F and A, C, such that

- $C(p) \in L_n$, *i.e.*, $(F \cup A, C)$ is a "compiler" under Definition 3.3, and
- $T_{\text{perf}}\left(C(p)\right)$ is maximized subject to the constraint $T_{\text{correct}}\left(C(p)\right) = 1$

The set of possible compositions of compiler and LLM-based components is infinite; moreover, each level-specific LLM-based generation is in general nondeterministic. It is therefore impractical to explore the entire search space, and we design an LLM agent to approximate a solution to the optimization problem. In the next section, we describe the design of our multi-agent system.

3.2 Architecture

In order to orchestrate the lowerings and rewrites performed by both existing compiler components and LLM-based level-specific agents, we develop the multi-agent system shown in Figure 3—the organizational structure is similar to the planner-executor model of Erdogan et al. [13]. The guiding agent, also an LLM, is empowered to make tool calls [39] to components of the existing compiler and to level-specific optimization agents at each level of abstraction. When called, a level-specific agent performs a rewrite of the input program at its particular level of abstraction, and returns the optimized program to the guiding agent for further calls. In the next subsections, we describe each of these agents, what context they have access to, and how they communicate with each other.

3.3 Guiding Agent

The purpose of the guiding agent is to coordinate the actions of each level-specific agent and to interleave calls to these agents with calls to existing compiler components. We therefore design the guiding agent to have access to six "tools" [39] or function calls: three corresponding to the existing components of the LLVM compiler, and three level-specific LLM-based optimization agents. Once started, the guiding agent proceeds in a standard tool calling loop. First, it makes an inference call that determines, based on current context, which of the available tools to call. It then forms a call to the selected tool in appropriate syntax (in practice, a JSON string), and calls the selected

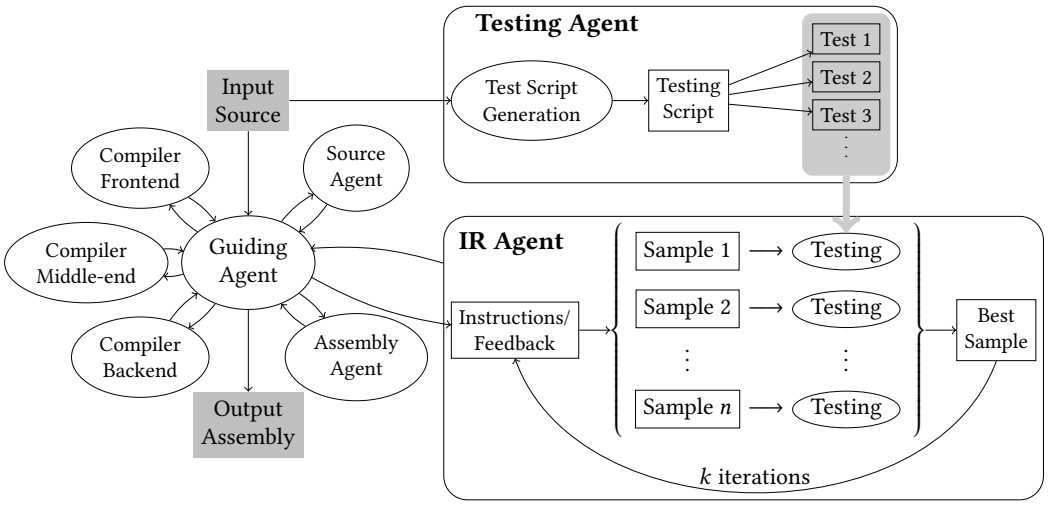


Fig. 3. Our multi-agent architecture, showing the guiding agent, testing agent, and level-specific optimization agents. The source and assembly agents follow the same design as the expanded IR agent.

tool. Once the tool has finished running, it returns, providing a string that gives feedback to the guiding agent. As we describe in the next section, this feedback from the level-specific agents includes performance and correctness measurements for the generated code. The feedback string is appended to the guiding agent's context, and the LLM is called again to select the next tool to call. Within the parameters of the provided tools, our design philosophy is to give the guiding agent maximum freedom to chose among the available tools.

There is one variable parameter which constrains the guiding agent, b, representing its compute budget. During a run of ACCLAIM, the guiding agent's context includes a budget tracker, which is updated after each tool call. We define a call to any level-specific agent to have unit cost, while calls to existing compiler components have cost 0. Compiler components are free to use since, unlike level-specific agents which make multiple LLM inference calls, existing compiler components run locally on a CPU and have comparably small runtime. We thus enforce no limit on the number of calls to existing compiler components; however, the guiding agent's prompt directs it to make calls to level-specific optimization agents, which in practice prevents runaway calls to the compiler. The guiding agent's prompt also describes the budget limitation, allowing the LLM to reason about how to divide the budget most effectively among level-specific optimization agents. We enforce no architectural limits on the order of calls to level-specific agents, allowing repeated calls and backtracking. Indeed, the guiding agent may make repeated calls to a particular level-specific agent if it is unsatisfied with the results achieved—we evaluate the choices it makes in Section 4.4.

Once the budget is exhausted, ACCLAIM stops calls to the guiding agent LLM and reports the end results to the user. If assembly level has not been reached (*i.e.*, the run of ACCLAIM has not fulfilled Definition 3.3), then the best-performing source or IR program produced so far is compiled with the existing compiler components and returned to the user as output. Intermediate optimization results from each call to a level-specific agent are also logged for later analysis.

3.4 Level-Specific Agents

The responsibility of a level-specific optimization agent is to take as input code at a particular level of abstraction and output new code at that same level of abstraction which is correct and

performant. Each level-specific agent is provided as context the input program and a string given by the guiding agent describing possible optimizations to attempt. After a run, it returns the optimized code along with a performance measurement to the guiding agent. In practice, ACCLAIM includes three level-specific agents: one each for C source code, LLVM IR, and x86 assembly. However, the design of the level-specific agents is not particularized to any one level of abstraction; we design a general level-specific agent and instantiate it once for each level, varying prompts and context only be replacing the name and description of the given level of abstraction. The design allows for easy generalization of ACCLAIM in case additional or different levels of abstraction are required.

A level-specific optimization agent could in principle simply call an LLM with its provided context; however, we include two classical enhancements for level-specific agents [3]: parallel sampling and iterative refinement feedback loops [30]. To that end, each level-specific agent is parameterized by two values:

- k, the number of feedback loop iterations performed during a run of the agent, and
- n, the number of generations collected during each pass through the feedback loop.

A level-specific agent then works as follows. First, it makes n calls to the LLM with identical context, producing a sample with n optimized program variants. It then calls the testing agent on each of these generated programs, yielding both performance and correctness results. The best-performing correct generated program is selected and replaces the agent's input program, and the loop is repeated k times, selecting the best-performing generation each time. We chose to perform sampling within each loop iteration rather than iterative refinement on each sample to limit computational resource use in practice. If a sample contains no correct programs, an aggregate of the feedback from the testing agent on the programs in the sample is appended to the level-specific agent's context and used to guide future optimization. The feedback includes compiler error messages, in the case that the generated program cannot be compiled to run, or a list of failing test cases, when the programs can be run but do not produce the correct results. After k iterations of iterative refinement, calls to the level-specific agent's LLM are stopped and the best-performing program so far is returned to the guiding agent along with the measured performance. If no successful programs were generated, correctness feedback is returned to the guiding agent instead.

We construct the level-specific agents to be modular and compatible with a large number of performance improvement strategies. For instance, as we discuss in Section 6, the level-specific agents could be improved with RL, ICL, or other enhancements as proposed by Wei et al. [49]. However, the extent of the practical benefits of these techniques is largely orthogonal to our contribution; our framework supports any choice of model, learning, and reinforcement strategies for level-specific agents. We practically implement two basic enhancements, but leave to future work the question of which enhancements and configurations are best suited to each class of inputs.

3.5 Testing Agent

In addition to the compiler and level-specific LLM-based optimization tools available to the guiding agent, we also develop a test generation agent whose responsibility is to generate test cases and run input programs to measure both performance and correctness. Design of the testing agent faces two important challenges. First, to adequately test performance, it must run an input program on test cases which are sufficiently numerous and diverse to exercise all possible behaviors of the program. Second, to fully measure performance, it must use test cases with a wide enough range to demonstrate the behavior of the program. We address both problems by constructing the testing agent to generate not individual test inputs, but rather a script which generates inputs in a deterministic manner.

We therefore split the testing agent into two phases, one that runs based on static information at the start of a run of ACCLAIM, and another which runs each time the testing agent is invoked by a level-specific agent, as shown in Figure 3. At the start of a run of ACCLAIM, the testing agent takes as context the initial input program and generates a script that can produce individual test inputs on demand. We divide the inputs produced by the testing script into two types: *correctness exploration inputs* which are designed to probe the correctness of a generated program and, for instance, test edge cases; and *large scale inputs*, which are designed to span a large range of input sizes to properly evaluate the runtime of the program under test. We define the test script generation task so that the testing script takes as input a parameter i which determines the size of the large scale inputs. For each call to the testing agent, we generate C correctness exploration inputs, and a further L large scale inputs spanning multiple orders of magnitude.

In the second phase, which occurs every time a level-specific agent calls the testing agent, the script is run to generate $C + L$ inputs to the provided program; in our experiments, we have $C = 10$ and $L = 5$. The testing agent compiles the program into an executable from whatever level of abstraction was provided, and runs the program on each of the inputs, measuring the running time and comparing the program's output to the output from the reference initial input program, measuring both correctness and performance across all of the correctness exploration and large scale inputs. It then reports the performance and correctness results to the level-specific agent, including any compiler error messages encountered during the test.

Because different input programs have different purposes and take different categories of input (integers, floating-point numbers, strings, grids of data, *etc.*), the contents of the test-generation script are highly program-dependent. The correctness exploration inputs are chosen during test script generation to explore as many possible program behaviors as possible. The large scale inputs, on the other hand, are designed to have size in proportion to the input parameter i—if i increases by an order of magnitude, then the size of the input to the program under test should also increase by an order of magnitude. However, the meaning of input size differs for each program. For example, the testing script for a program which calculates the square of a number could just return i. Another program, like that in Example 4.1, might expect a grid as input, and the test script could produce a random $i \times i$ grid. Still other programs could expect negative values as input, and have running time that grows as the input *shrinks* in magnitude. We therefore rely on the LLM's reasoning ability to produce a script that generates inputs which grow as i grows. Having inputs that can span orders of magnitude is critical to accurately measuring performance, as two programs with vastly different asymptotic performance may both appear to run quickly on a set of small inputs.

4 Evaluation

We evaluate our multi-level optimization method on a benchmark set of C programs, ablating to compare different configurations of ACCLAIM and analyzing optimization traces to help explain how multi-level optimization achieves speedups. We summarize our key results as follows:

- Multi-level optimization with ACCLAIM outperforms optimization at any single level of abstraction, and a naive multi-level baseline.
- The guiding LLM regularly calls all level-specific agents and compiler components, and often repeats calls or backtracks to improve performance. Source-level optimization contributes the most to speedups.
- Performance improves as ACCLAIM is given a higher compute budget, and resources are better devoted to iterative refinement than parallel sampling.

Problems		
Original Dataset	3,365	100%
No Programs Compile	199	5.9%
Test Generation Failed	205	6.1%
Less Than 10 Programs	1,064	31.6%
Low Running Time Variance	1,333	39.6%
Remaining Problems	564	16.7%

(a) Breakdown of problems in the CodeNet dataset. Each problem contains several solution programs.

Programs			
Original Dataset		754,058	100%
Compiler Error		141,740	18.8%
Compiler Warning		64,775	8.6%
Compiler Success	No Successful Test Cases	15,969	2.1%
	Remaining Programs	547,543	70.5%

(b) Breakdown of program filtering across all problems in the CodeNet dataset. We exclude the approximately 30% of programs that do not compile in original form.

Table 1. Dataset filtering by problems and programs.

4.1 Dataset

We use as benchmarks C programs taken from the Project CodeNet dataset [36], which have been used to evaluate LLM-based optimization techniques in the past [40, 49]. The dataset consists of 4,053 competitive programming *problems*, with multiple *programs* solving each one. We restrict our attention to C programs, for which 3,365 problems have at least one program; the total number of programs is 754,058. Many of these programs contain errors, do not compile, or are too small to provide meaningful performance measurements. A secondary contribution of our work is therefore a set of benchmarks useful for LLM-based performance improvement evaluation. We remove the approximately 30% of programs in the dataset that do not compile in original form, as shown in Table 1b. We further exclude 205 problems, shown in Table 1a, for which our testing agent fails to produce testing scripts, equating to 15,969 programs. To fairly test our agent on programs where opportunities for optimization exist, we use variability across the solutions to a problem as a proxy for the existence of optimization opportunities [40]. For any problem with at least 10 programs, we sample from problems where the difference in running time between the slowest and fastest solutions is at least 10%. This results in 564 problems, and we randomly sample 10 programs from each, giving a dataset of 5,640 programs with opportunities for optimization. We perform ablation studies and comparisons using a random sample of 100 programs to make experiments at scale tractable. Random sampling is in line with prior work using a 200-program sample [49].

4.2 Experimental Setup

ACCLAIM is is implemented in the Strands Framework [28] and made publicly available on Github[2]. It includes a central guiding agent that is given access to the core agents and `clang` components as tools. We use LLVM version 18.1.3, and measure speedups against the performance of the original source code compiled with `clang -O3`. We perform experiments on a machine with 96 Intel Xeon Platinum 8275CL CPUs and 1TB RAM, accessing Claude 3.7 Sonnet via Amazon Bedrock. Each agent run is limited to one hour.

[2]https://github.com/amazon-science/acclaim.

We design our implementation and experiments to investigate three research questions about compiler-LLM cooperation:

- **RQ1: Effectiveness.** Does optimization at multiple levels outperform single-level LLM-based optimization?
- **RQ2: Process.** How does compiler-LLM cooperation achieve performance improvements, and what choices does the guiding agent make?
- **RQ3: Design.** What is the optimal configuration for multi-level LLM-based optimization?

4.3 RQ1: Effectiveness

The first research question asks whether multi-level optimization performs better than optimization at a single level of abstraction. To answer this question, we fix the total computational budget and perform experiments comparing ACCLAIM against four baselines:

(1) An agent restricted to performing source code optimizations only
(2) An agent restricted to performing IR optimizations only
(3) An agent restricted to performing assembly optimizations only
(4) A per-input single-level portfolio of the above three approaches

Comparisons 1–3 investigate the performance of multi-level optimization relative to any particular choice of abstraction level. Comparison 3 is a proxy for comparison against the approach of Wei et al. [49], for which an open source implementation is not currently available.[3] Comparison 3 takes the same fixed computation budget used for all the other comparisons, and divides it evenly among the source, IR, and assembly levels. It then performs level-specific optimization for each of these levels in parallel and at the end selects the best-performing optimization on a per-program basis. Comparison 1 is similar to source-only approaches like that of Gao et al. [15]. To our knowledge, no existing work has directly performed LLM-based optimization at the IR level in the CPU context; the closest analogue to Comparison 2 is the approach of Grubisic et al. [17], which optimizes code size and predicts output LLVM IR.

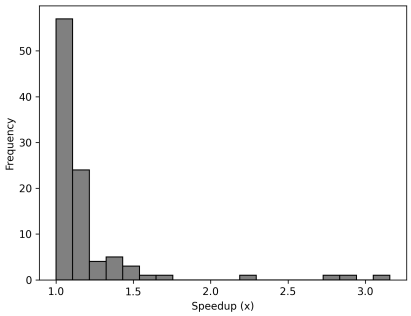
The results are shown in Table 2, which gives the geometric mean speedup for each configuration, along with the speedups at each percentile. ACCLAIM outperforms all of the baselines, though the source-only agent also performs well. For each case, Table 2 shows that speedups are heavily clustered at the top of the distribution; speedups for the 75th and especially 99th percentile are far higher than the mean. The top-percentile speedups demonstrate that, at least for some programs, LLMs are capable of identifying optimizations at each level of abstraction.

The results suggest that LLM-based optimization in general, and our multi-level approach in particular, achieves large speedups for some programs, while not improving others. Figure 4 shows the distribution of speedups for each configuration. LLM-based optimization at any level of abstraction does not substantially speed up most programs, with clusters of no or negligible speedup around 1.0× in Figures 4b–4d, but a smaller number of programs sped up by much more—the absolute highest speedups, though, vary with each level, as shown by the differing x-axis scales. ACCLAIM follows the same trend, with very little improvement for most programs, but several programs are sped up by more than 50%. The trend in the data arises because most programs do not contain many opportunities for optimization that existing compilers do not exploit. It is only rare programs where an LLM is able to exploit a high level optimization, which then yields a high speedup. The phenomenon of infrequent but large speedups during LLM-based optimization was also observed by Wei et al. [49], which found a much higher average speedup for the top percentiles of programs.

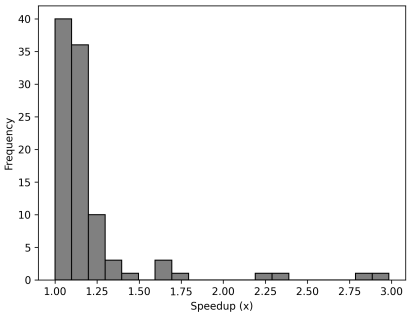
[3] Exact speedups, though, are not directly comparable with the approach of Wei et al. [49] because of hardware differences and the lack of RL in our level-specific agent.

Table 2. Speedups (×) achieved by our method and level-specific baselines.

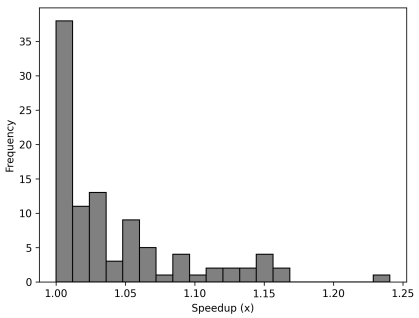
Configuration	Geo. Mean	25th	50th	75th	99th
Source-only (1)	1.18	1.06	1.12	1.18	2.98
IR-only (2)	1.04	1.00	1.02	1.06	1.24
Assembly-only (3)	1.03	1.00	1.01	1.05	1.16
Single-level portfolio (4)	1.02	1.00	1.00	1.02	1.20
ACCLAIM	1.25	1.04	1.10	1.16	16.97



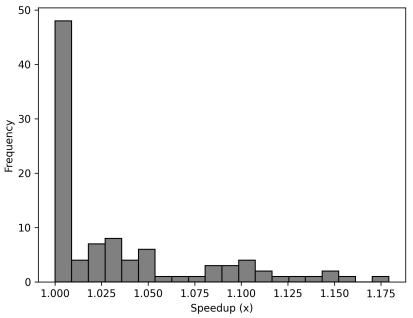
(a) ACCLAIM*.



(b) Source-only.



(c) IR-only.



(d) Assembly-only.

Fig. 4. Histogram of speedups produced by ACCLAIM and each level-specific baseline with $b = 18, n = 2, k = 2$, scaled to show distribution of speedups. *Figure 4a omits one program with speedup 1384× for clarity; see the example.

Example 4.1. The most striking example of high speedups for few programs is perhaps benchmark `p02644_s879314742`; this program is a solution to a competitive programming problem involving path-finding on a square grid, where certain squares are blocked. The program reads as input the contents of the `W` × `H` grid, the maximum movement in a single step `K`, and the source and target coordinates. It then performs a breadth-first search through the grid to find the best path. ACCLAIM speeds up the program by 1384× on overage over a set of 10 inputs with grid size between 5×5 and 1000×1000—the testing agent (Section 3.5) is used to generate inputs spanning several orders of magnitude to fully assess the program's performance.

The performance improvements occur primarily at the source level, and span the program; we here highlight one important improvement to demonstrate the types of optimizations an LLM uses

```
c = (char**) malloc(sizeof(char*) * (H + 2));
for (i = 1; i <= H; i++) {
    c[i] = (char*) malloc(sizeof(char) * (W + 2));
    scanf("%s", &(c[i][1]));
}
c[0] = (char*) malloc(sizeof(char) * (W + 2));
c[H+1] = (char*) malloc(sizeof(char) * (W + 2));
```

(a) Original program snipped that calls `malloc` $O(H)$ times.

```
char **grid = (char**) malloc((H + 2) * sizeof(char*));
char *gridData = (char*) malloc((H + 2) * (W + 2) * sizeof(char));
memset(gridData, '@', (H + 2) * (W + 2)); // Initialize all to walls
```

(b) Program after source-level optimization by ACCLAIM, calling `malloc` only twice.

Fig. 5. Source-level optimizations performed by ACCLAIM on benchmark `p02644_s879314742`.

Table 3. Speedup contributions, number of calls, proportion of individual samples correct, and proportion of samples that contain at least one correct solution. Shown for to each level-specific agent during a run of ACCLAIM, with $b = 18, n = 2, k = 2$. Speedup contributions do not add up to the overall speedup since they are multiplicative, rather than additive.

Level	Speedup Contribution	# of Calls	% Correct Generations	% Correct Samples
Source	1.222×	427	53.5%	84.8%
IR	1.012×	151	45.4%	41.0%
Assembly	1.010×	226	48.3%	57.4%
Overall	1.248×	804	51.4%	69.4%

to outperform existing compilers. Figure 5a shows a snippet of the program that allocates memory to hold the input grid; the program first allocates space for $H + 2$ pointers, one for each row of the grid. It then separately allocates memory for each grid row in a loop, requesting memory from the operating system H times, and then finally for the first and last rows which represent boundaries. ACCLAIM optimizes the program, producing the code shown in Figure 5b. Rather than calling `malloc` in a loop, the optimized program allocates space for the entire grid, $(H + 2) \times (W + 2)$ cells, in one go, asymptotically reducing the number of highly expensive system calls to allocate memory from linear in H to constant. Adjustments to other parts of the program, including simplifications of the BFS implementation, also contribute to the overall speedup.

4.4 RQ2: Process

For RQ2, we investigate how compiler-LLM cooperation works, analyzing the optimization logs it produces and quantifying the decisions made by the guiding agent.

Level Contributions. Table 3 shows the frequency with which the guiding agent calls each level-specific agent and the speedup contribution of each. The speedup contributions show that source level optimization is the most effective, matching the results in Table 2, while IR and assembly optimization levels contribute far less to optimization. However, the contributions from each level are neither independent nor cleanly distinguishable. Interactions between different levels occur both as synergies in optimization (*i.e.*, an optimization at one level enables or prevents an optimization

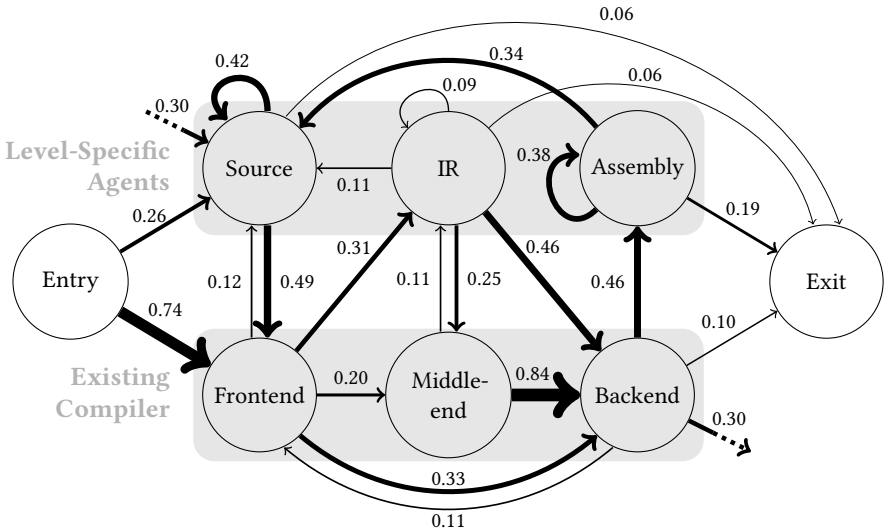


Fig. 6. Graphical representation of ACCLAIM's transition frequencies as an automaton. Arrow width is proportional to frequency, and transitions with frequency less than 0.05 have been omitted for clarity. There is a single transition from the compiler backend to source-level optimization with frequency 0.30 represented by disconnected dotted arrows.

at another level) and as budget-use choices made by the guiding agent (*i.e.*, calling one level-specific agent because another was ineffective). These effects result in the difference difference between the level-specific *contributions* shown in Table 2 and the *level-only baseline* results in Table 2.

Column "% Correct Generations" of Table 3 gives the proportion of individual LLM calls that pass all test cases—each of the n members of the sample taken during level-specific optimization are counted independently. Column "% Correct Samples" gives the percentage of level-specific agent samples *in which at least one generated program was correct*—this is a "pass@n" metric. The proportion of correct generations is a reflection of the model's ability to produce correct code, while the proportion of correct samples measures the benefit of sampling n times.

Since the guiding agent discards incorrect generations (*i.e.*, they contribute no speedup), rates of incorrect generation are an important factor in defining each level's speedup contribution—and indeed, rates of correct generation are lower at IR and assembly level than for source code. However, the real difference lies in the number of level-specific agent runs where at least one of the n samples is correct: this is 85% for source but only 41% for IR and 57% for assembly. These results suggest that the IR and assembly agents are more likely to produce "all-or-nothing" samples, while at source code level, sampling produces greater variation.

Agent Behavior. We also investigate the choices made by the guiding agent during runs of ACCLAIM. Table 3 shows that the guiding agent's calling behavior mirrors the level-specific agents' correctness: it calls source-level optimization at almost twice the rate of the other levels. In addition to the level-specific LLM-based optimization agents, the guiding agent also has access to the existing components of clang—the frontend, middle-end, and backend. Figure 6 shows the outbound transition frequency with which the agent transitions between each of the six tools it has access to. The most frequent transitions for the source and IR level agents are to the compiler frontend and

```
%10 = add nsw i32 %8, %9
%11 = sdiv i32 %6, %10
%12 = mul nsw i32 %11, 7
%13 = srem i32 %6, %10
%14 = icmp slt i32 %13, %8
```

(a) Initial LLVM IR program before optimization.

```
%10 = add nsw i32 %8, %9
; Combine division and multiplication operations
%11 = sdiv i32 %6, %10
%12 = shl nsw i32 %11, 3
%13 = sub nsw i32 %12, %11
%14 = srem i32 %6, %10
%15 = icmp slt i32 %14, %8
```

(b) Program after optimization by the IR level-specific agent. The comment marked with ";" was inserted by the agent.

Fig. 7. Exceprt of IR code during optimization of benchmark `p00575_s033386858`.

backend, respectively; this shows the cooperation between LLM agents and the compiler enabled by ACCLAIM.

The initial transition frequencies (26% for source-level optimization and 74% for the compiler frontend) suggest that the guiding agent is not simply calling LLM-based source-level optimization, but instead producing intermediate results with which to generate feedback. The agent also frequently re-runs level specific agents—42% of the time for source, 9% for IR, and 38% for assembly. This tendency may indicate that, when results produced by a level-specific tool are not satisfactory, the guiding agent calls the level-specific tool to "try again". One surprising trend is the frequency of transitions from the compiler backend to both the compiler frontend (11% of the time) and source-level optimization (30% of the time). This indicates an optimization workflow in which the guiding agent has produced a full optimization but then starts over from the highest level of abstraction.

Example 4.2. To illustrate how compiler-LLM optimization produces speedups across multiple levels, we show the example of benchmark `p00575_s033386858`. This program models a game of consecutive days: for each day an action is performed, a player earns a points, but b points are earned after 7 consecutive days. The program takes as input a target number of points c and computes the number of days needed to earn c points. Figure 8 shows the performance of the input program at each intermediate point during a run of ACCLAIM; gray lines indicate unselected generations during level-specific agent sampling. Samples are not selected either because they have worse performance than that already achieved, or because they are not correct. Performance improvements are achieved at both the source and IR levels, with the primary contribution coming from an IR transformation.

Figure 7 shows an excerpt of the program during the second round of IR-level optimization. The IR agent converts a multiplication by the constant 7 (Figure 7a) into the two instruction sequence shift left by 3, equivalent to multiplication by 8, and then subtract 1 (Figure 7b). LLVM is capable of performing this optimization, but for the subject program does not do so, even at `O3`; the optimization is also not performed during the compiler backend. In this instance, LLVM's cost model, which is designed to be general, mispredicts the cost of the multiplication operation and chooses not to replace it with a shift instruction; the LLM-based IR agent does so instead, leading to a speedup increase from 1.06× to 1.16×.

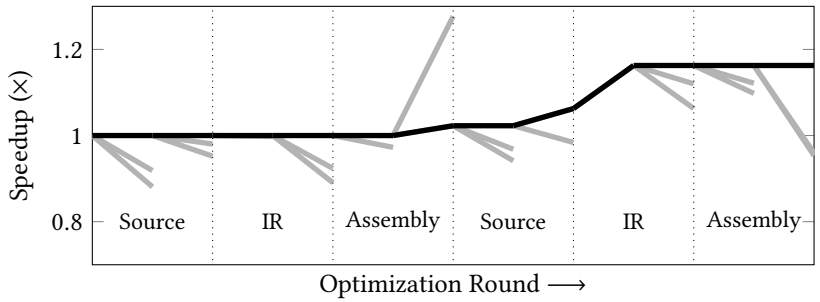


Fig. 8. Trace of optimization for benchmark `p00575_s033386858`, with $b = 6$, $n = 2$, $k = 2$. Gray lines show samples within each level-specific loop, including incorrect samples; black lines show the selected best correct optimized version.

4.5 RQ3: Design

To investigate the best design of a compiler-LLM cooperation system, we perform several ablations to study the impact of changes in compute budget and configuration settings on the performance of ACCLAIM. We consider how compute budget should be distributed, how the performance of compiler-LLM cooperation depends on overall compute budget, and which LLMs are best-suited to the task.

Model Selection. We evaluate compiler-LLM cooperation with six language models available as of August 2025, with the results shown in Table 4. As in Table 3, "% Correct Generations" is the proportion of individual LLM calls that produce a correct program, while "% Correct Samples" gives the number of samples with at least one correct generation. We evaluate with Claude 3.7 Sonnet, in addition to five open-source models: two variants of Llamma (`llama4-maverick-17b-instruct` and `llama3-3-70b-instruct`), Qwen 2.5, and two different sizes of Qwen 3 models (4b and 30b, with mixture-of-experts). The results show that Claude 3.7 Sonnet outperforms all of the open-source models. Most of the difference is attributable to differences in the number of correct generations—since each incorrect generation results in a sample being discarded, and thus no speedup, models that only rarely produce correct samples cannot achieve any speedup. By investigating the optimization traces, we find that one reason for the poor correctness performance of the Qwen and Llama models is related to their tool-calling ability rather than their ability to generate code; indeed, many calls to level-specific agents fail because the guiding agent is unable to generate well-formed tool calls and outputs plain text calls which are ineffective. Tool-calling is an integral part of the compiler-LLM cooperation framework, though it may be possible to add an additional tool-calling harness [20] or to improve models' compiler-related tool calling ability through further context engineering or finetuning with in-context examples [38].

Distribution of Compute Budget. The design of ACCLAIM includes three tunable parameters for computation budget: b, the number of level-specific calls the guiding agent can make, n, the number of parallel samples collected in each loop of a level-specific agent, and k, the number of feedback loop iterations performed by each level-specific agent. The computational cost of running experiments prevents a full scan of values of b, n, and k, so we compare three versions of ACCLAIM with fixed b and cap the quantity $n \times k$; the results are shown in Table 4. We find that, using the most successful model (Claude 3.7 Sonnet), devoting more computational resources to feedback loops instead of parallel sampling ($n = 1, k = 4$) produces the best results. An even division between sampling and feedback loops ($n = 2, k = 2$) is not far behind, while devoting all resources to parallel sampling instead of feedback loops performs worse (1.11× speedup). Iterative refinement with the feedback loop helps the level-specific agents reason about and correct their mistakes; in the

Table 4. Results for ablation studies of different configurations of ACCLAIM.

Model	b	n	k	Speedup	% Correct Generations	% Correct Samples
`claude-3.7-sonnet`	18	2	2	1.25	51.4%	69.4%
`claude-3.7-sonnet`	6	1	4	1.19	54.5%	64.4%
`claude-3.7-sonnet`	6	2	2	1.16	48.0%	48.1%
`claude-3.7-sonnet`	6	4	1	1.11	42.1%	48.4%
`llamma4-maverick-17b-instruct`	6	2	2	1.01	13.0%	31.9%
`llama3-3-70b-instruct`	6	2	2	1.00	0%	0%
`qwen2.5-coder-7b`	6	2	2	1.00	0%	0%
`qwen3-4b`	6	2	2	1.00	1.3%	16.8%
`qwen3-30b-a3b`	6	2	2	1.04	8.3%	27.7%

example above, for instance, after receiving failed test results, the IR-specific agent proceeds to the next iterative refinement loop with feedback indicating what to do next: "Consider using register hints (like register variables) and further simplify the conditional logic".

The measurements of the number of correct generations and samples shed some light on the reason for the results: the proportion of correct generations is highest for $k = 4$, suggesting that the level-specific agent's feedback loop increases the frequency with which the model produces correct code. While $n = 4$ (a larger sample size) results in a larger spread between the number of correct generations and number of correct samples, the decrease in feedback loops reduces the absolute number of correct generations, and thereby the overall speedup.

Compute Time Scaling. To test whether increasing overall computation budget improves model results, we triple the overall compute budget and perform experiments with the best-performing model and parameters $b = 18$, $n = 2$, $k = 2$; we keep n and k equal to eliminate any effects from changing the distribution of compute budget between n and k. The results show that increasing the compute budget improves overall performance, increasing mean speedups from 1.19× to 1.25×. This result suggests that the performance improvements achieved by compiler-LLM cooperation scale with increases in compute budget, and that this increase has not yet leveled off by $n = 18$. Whether it is worthwhile to invest greater resources to achieve small performance improvements likely depends on the tradeoffs of a particular application domain. Since some compiled programs may be run billions of times by millions of users, even small incremental improvements to runtime could merit a high inference cost.

5 Discussion and Future Work

5.1 Generalization to Other Compilers

We realize ACCLAIM using the LLVM framework [24]; LLVM is the most commonly-used compiler in prior work [11, 17, 29]. However, the structure of compiler-LLM cooperation is general, and could be applied to other compilers or computing environments. In the CPU context, GCC has the same frontend, middle-end, backend structure as clang and its own IR [42]; ACCLAIM could be adapted to cooperate with GCC with minimal engineering effort. Other source languages, like Rust, could also be targeted. Extend compiler-LLM cooperation to software targeting GPUs is also of great practical importance. As in the CPU context, the GPU software stack includes several levels of abstraction: source code may be written in PyTorch, then compiled to the Triton language [45]. Triton kernels are then lowered to MLIR and then to LLVM IR where they benefit from the same LLVM optimization passes used for CPU programs. Finally, they are lowered to a low-level hardware-specific ISA

like AMD GCN or Nvidia PTX. Existing work has already proposed level-specific rewrites for PyTorch [4, 34] and Triton [14], so future work could integrate these level-specific optimization strategies with the existing components of the Triton and LLVM compilers to bring the benefits of compiler-LLM cooperation to GPU kernels as well—optimization of GPU kernels is of particular interest to improve the performance of LLMs themselves.

Compiler-LLM cooperation may also have applications in less closely-related applications. For instance, in Definitions 3.1–3.3, we state lowerings and rewrites as the atomic operations of compilers, but recently, interest has also turned to inverse lowering, *i.e.*, lifting [47], including by using LLMs [43]. Future work could address whether cooperation between existing compiler components and verification tools like ACCLAIM may be able to improve the correctness of LLM-based decompilations.

5.2 Correctness and Verification

Our use of testing to check the correctness of LLM-generated programs is in line with existing work in both the CPU [15, 40, 49] and GPU [4, 34] use cases. However, testing is not infallible, and may miss bugs or correctness issues in the LLM-generated code at any of the three levels of abstraction we address. We mitigate the risk of incorrect code generation through the design of the testing agent (3.5), which can produce additional test inputs on demand and includes test cases designed to probe the correctness of the generated code. The structure of compiler-LLM cooperation itself helps to ensure correctness by combining the guaranteed correctness of existing compiler optimizations[4] with the possibly error-bearing optimization potential of LLM-based code generation. The design of the testing agent also does not share full test cases with the level-specific optimization agents in order to reduce the likelihood of reward hacking [41]; we have not qualitatively observed reward hacking behavior in the reasoning traces from runs of ACCLAIM.

Nevertheless, future work could address the problem of fully verifying the correctness of output programs from compiler-LLM cooperation. Such verification could take the form of translation validation [32], fully checking the equivalence of the input and output programs. Existing tools like Alive2 [29] may be useful for the IR portions of this task, while more sophisticated approaches would be required to verify the equivalence of input C and output assembly programs. Better translation validation methodologies would increase confidence in the programs produced by ACCLAIM, but their development is largely orthogonal to the strategy of compiler-LLM cooperation; the strategy is modular and would be compatible with the drop-in replacement of a testing agent with a verification agent instead. A verification agent which performs translation validation may even be able to provide more granular feedback to the LLM during runs of a level-specific agent.

5.3 Computational Cost

One important drawback of compiler-LLM cooperation is high computational cost: a run on a single program requires $b \times n \times k$ inference calls to a model, each with a number of tokens proportional to input program size. The results of Section 4.5 show that increasing compute budget can increase the performance gains produced by ACCLAIM. Future work may determine where the point of diminishing return lies. This point likely depends heavily on input program features, the selected model, and the performance of the existing compiler components. The use case of the input program matters too: for a program that goes on to wide use as infrastructure software, devoting significant computing resources to achieve a small speedup may be practically worthwhile, while a program that will only be run one time by a single user may not merit the cost of LLM inference at all.

[4]Compilers perform correct transformations, modulo compiler bugs. We expect that compiler bugs affecting correctness are vanishingly rare relative to the frequency of incorrect LLM-generated code.

Table 5. Summary of related work on LLM-based code optimization.

	GPU			CPU			
Approach	Ouyang et al. [34]	Kernel-LLM [14]	Kevin [4]	SBLLM [15]	Shypula et al. [40]	Super-Coder [49]	ACCLAIM
Source Optimization	✓			✓	✓		✓
Intermediate Optimization		✓					✓
Low-level Optimization			✓			✓	✓
Repeated Sampling	✓		✓	✓	✓		✓
Iterative Refinement	✓		✓	✓			✓
Reinforcement Learning		✓	✓			✓	
Supervised Fine-Tuning		✓			✓		

Future work could also explore methods of reducing the computational burden associated with compiler-LLM cooperation. The modular design of ACCLAIM allows mixing and matching of models—a larger, more expensive model could be used for the guiding agent, for instance, while a smaller, possibly finetuned, model is used for each level-specific agent. Computational burden could also be reduced by focusing LLM-based optimization only to parts of the input program; indeed, the guiding agent could take on the additional task of selecting portions of the input program which are most likely to be amenable to LLM-based optimization at the level of functions, basic blocks, or even individual instructions. This division may also boost correctness by decreasing the "attack surface" on which LLM-based code generation can introduce bugs.

6 Related Work

LLM-based Code Optimization. Our approach is most closely related to the body of work on LLM-based code optimization. Existing approaches in this area have used LLMs to optimize source code; SBLLM [15] integrates LLM-generated re-writes with an evolutionary search for better-performing code, while Shypula et al. [40] train an LLM with supervised fine-tuning and self-play to optimize C source code. SuperCoder [49] performs the same task at the level of x86 assembly; it employs reinforcement learning to train LLMs to generate fast and correct assembly code, given as input C source code and compiler-generated assembly. Constrained decoding [12, 35] has been proposed as a method to boost the correctness of generated code by limiting the output of the LLM to match the grammar of the given language. LLMs have also been employed to optimize programs in the GPU context. Here, the seminal work is KernelBench [34], which collects a dataset and establishes a baseline for LLM-driven generation of GPU kernels. It operates at the level of PyTorch, but permits the LLM to insert lower-level kernel code as well, and uses both repeated sampling and iterative refinement with feedback derived from compilation and testing of the generated kernels. Other work has improved upon the base of KernelBench [9, 31]. KernelLLM [14] optimizes code in the Triton language, an intermediate between PyTorch and low-level CUDA kernels and takes advantage of both reinforcement learning and supervised fine-tuning. The state-of-the-art approach is Kevin [4], which uses multi-turn reinforcement learning to further improve performance on the KernelBench dataset.

Table 5 summarizes the relationship between these approaches. ACCLAIM operates on CPU-targeted programs and implements repeated sampling and iterative refinement with a feedback loop as do existing approaches. Our approach differs in that it integrates optimization at multiple levels of abstraction into a single optimization workflow, and includes a guiding agent to orchestrate the process. ACCLAIM provides a proof of concept realization of compiler-LLM cooperation, including the two most common performance improvements in the existing literature: repeated sampling and iterative refinement. We leave to future work the question of how much practical improvement could be gained by incorporating other more advanced improvements like reinforcement learning and supervised fine-tuning, noting only that the design of our level-specific agents would be compatible with both.

Integration of Compilers and LLMs. In addition to direct LLM-based optimization of input programs, existing work has harnessed LLMs to direct existing compiler passes and settings. CompileAgent [19] gives existing compilers and other build tools as tools to an LLM agent, and constructs a feedback loop by passing compiler error messages back to the LLM. LLMCompiler [11] performs model training to enhance understanding of programs at IR level, and uses the results to chose optimization passes and directly reduce code size [17]. Tang et al. [44] use LLMs to aid in searching the space of existing compiler optimizations in the GPU context. Our approach shares the integration of LLM-based reasoning with existing compiler optimizations, but differs in that it incorporates calls to compiler components with direct LLM code generation. We also allow the guiding agent to chose between the front-, middle-, and back-end of a compiler, rather than focusing on middle-end optimization passes.

Compiler Phase Ordering. In exploring the space of possible rewrites and lowerings provided by existing LLVM and level-specific optimization agents, ACCLAIM's guiding agent confronts the well-known compiler phase ordering problem. Phase ordering followed soon after the first compiler with multiple phases [5, 46], grappling with the issues of both synergies and anti-synergies between optimizations as does ACCLAIM. Subsequent approaches developed methods for conducting an exhaustive search of optimization orders [22, 23] and for tailoring compiler phase choices to particular programs with iterative search [33, 37]. Phase ordering has also been tackled by pre-LLM machine learning approaches, including neural networks [2], reinforcement learning [1, 18, 27], and others [7, 8]. We address phase ordering by empowering an LLM-based guiding agent to chose between existing compiler-provided rewrites and LLM-based level-specific optimization agents.

7 Conclusion

This work has presented LLM-compiler cooperation, a novel approach to LLM-based code optimization which integrates LLM-based code generation with existing compiler components at three different levels of abstraction: source code, intermediate representation, and low-level assembly. LLM-compiler cooperation harnesses the "creativity" of LLM-based optimization but tempers it with the guaranteed correctness of an existing compiler. We realize LLM-compiler cooperation as ACCLAIM, which integrates calls to LLM-based optimization agents with existing compiler components. Our approach combines LLM-based optimization at multiple levels of abstraction with a guiding agent that coordinates use of LLM-based and existing compiler-based optimization techniques. Our extensive evaluation shows that compiler-LLM cooperation outperforms both existing compiler optimizations and LLM-based optimization at any single level of abstraction, achieving an overall speedup of 1.25× on average, with orders of magnitude improvement on individual input programs.

References

[1] Mohammed Almakki, Ayman Izzeldin, Qijing Huang, Ameer Haj Ali, and Chris Cummins. 2022. Autophase V2: Towards Function Level Phase Ordering Optimization. In *Machine Learning for Computer Architecture and Systems 2022*. https://openreview.net/forum?id=HEWao_L2IP_

[2] Amir H. Ashouri, Andrea Bignoli, Gianluca Palermo, Cristina Silvano, Sameer Kulkarni, and John Cavazos. 2017. MiCOMP: Mitigating the Compiler Phase-Ordering Problem Using Optimization Sub-Sequences and Machine Learning. *ACM Trans. Archit. Code Optim.* 14, 3, Article 29 (Sept. 2017), 28 pages. doi:10.1145/3124452

[3] Vidhisha Balachandran, Jingya Chen, Lingjiao Chen, Shivam Garg, Neel Joshi, Yash Lara, John Langford, Besmira Nushi, Vibhav Vineet, Yue Wu, and Safoora Yousefi. 2025. Inference-Time Scaling for Complex Tasks: Where We Stand and What Lies Ahead. arXiv:2504.00294 [cs.LG] https://arxiv.org/abs/2504.00294

[4] Carlo Baronio, Pietro Marsella, Ben Pan, and Silas Alberti. [n. d.]. Multi-Turn RL Training for CUDA Kernel Generation. https://cognition.ai/blog/kevin-32b.

[5] M. E. Benitez and J. W. Davidson. 1988. A portable global optimizer and linker. *SIGPLAN Not.* 23, 7 (June 1988), 329–338. doi:10.1145/960116.54023

[6] Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. 2024. Large Language Monkeys: Scaling Inference Compute with Repeated Sampling. arXiv:2407.21787 [cs.LG] https://arxiv.org/abs/2407.21787

[7] Stefano Cereda, Gianluca Palermo, Paolo Cremonesi, and Stefano Doni. 2020. A Collaborative Filtering Approach for the Automatic Tuning of Compiler Optimisations. In *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (London, United Kingdom) *(LCTES '20)*. Association for Computing Machinery, New York, NY, USA, 15–25. doi:10.1145/3372799.3394361

[8] Junjie Chen, Ningxin Xu, Peiqi Chen, and Hongyu Zhang. 2021. Efficient Compiler Autotuning via Bayesian Optimization. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 1198–1209. doi:10.1109/ICSE43902.2021.00110

[9] Terry Chen, Bing Xu, and Kirthi Devleker. 2025. Automating GPU Kernel Generation with DeepSeek-R1 and Inference Time Scaling. https://developer.nvidia.com/blog/automating-gpu-kernel-generation-with-deepseek-r1-and-inference-time-scaling/.

[10] Yongchao Chen, Yueying Liu, Junwei Zhou, Yilun Hao, Jingquan Wang, Yang Zhang, and Chuchu Fan. 2025. R1-Code-Interpreter: Training LLMs to Reason with Code via Supervised and Reinforcement Learning. *CoRR* abs/2505.21668 (2025). arXiv:2505.21668 doi:10.48550/ARXIV.2505.21668

[11] Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. 2025. LLM Compiler: Foundation Language Models for Compiler Optimization. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction* (Las Vegas, NV, USA) *(CC '25)*. Association for Computing Machinery, New York, NY, USA, 141–153. doi:10.1145/3708493.3712691

[12] Yixin Dong, Charlie F. Ruan, Yaxing Cai, Ruihang Lai, Ziyi Xu, Yilong Zhao, and Tianqi Chen. 2025. XGrammar: Flexible and Efficient Structured Generation Engine for Large Language Models. arXiv:2411.15100 [cs.CL] https://arxiv.org/abs/2411.15100

[13] Lutfi Eren Erdogan, Nicholas Lee, Sehoon Kim, Suhong Moon, Hiroki Furuta, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. 2025. Plan-and-Act: Improving Planning of Agents for Long-Horizon Tasks. *CoRR* abs/2503.09572 (2025). arXiv:2503.09572 doi:10.48550/ARXIV.2503.09572

[14] Zacharias V. Fisches, Sahan Paliskara, Simon Guo, Alex Zhang, Joe Spisak, Chris Cummins, Hugh Leather, Gabriel Synnaeve, Joe Isaacson, Aram Markosyan, and Mark Saroufim. 2025. KernelLLM: Making Kernel Development More Accessible. https://huggingface.co/facebook/KernelLLM.

[15] Shuzheng Gao, Cuiyun Gao, Wenchao Gu, and Michael R. Lyu. 2025. Search-Based LLMs for Code Optimization. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*. IEEE, 578–590. doi:10.1109/ICSE55347.2025.00021

[16] Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. 2024. RLEF: Grounding Code LLMs in Execution Feedback with Reinforcement Learning. *CoRR* abs/2410.02089 (2024). arXiv:2410.02089 doi:10.48550/ARXIV.2410.02089

[17] Dejan Grubisic, Chris Cummins, Volker Seeker, and Hugh Leather. 2024. Compiler generated feedback for Large Language Models. *CoRR* abs/2403.14714 (2024). arXiv:2403.14714 doi:10.48550/ARXIV.2403.14714

[18] Ameer Haj-Ali, Qijing (Jenny) Huang, John Xiang, William Moses, Krste Asanovic, John Wawrzynek, and Ion Stoica. 2020. AutoPhase: Juggling HLS Phase Orderings in Random Forests with Deep Reinforcement Learning. In *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), Vol. 2. 70–81. https://proceedings.mlsys.org/paper_files/paper/2020/file/5b47430e24a5a1f9fe21f0e8eb814131-Paper.pdf

[19] Li Hu, Guoqiang Chen, Xiuwei Shang, Shaoyin Cheng, Benlong Wu, Gangyang Li, Xu Zhu, Weiming Zhang, and Nenghai Yu. 2025. CompileAgent: Automated Real-World Repo-Level Compilation with Tool-Integrated LLM-based

Agent System. *CoRR* abs/2505.04254 (2025). arXiv:2505.04254 doi:10.48550/ARXIV.2505.04254

[20] Reid T. Johnson, Michelle D. Pain, and Jordan D. West. 2025. Natural Language Tools: A Natural Language Approach to Tool Calling In Large Language Agents. arXiv:2510.14453 [cs.CL] https://arxiv.org/abs/2510.14453

[21] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Yacine Jernite, Margaret Mitchell, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2023. The Stack: 3 TB of permissively licensed source code. *Trans. Mach. Learn. Res.* 2023 (2023). https://openreview.net/forum?id=pxpbTdUEpD

[22] Prasad A. Kulkarni, David B. Whalley, Gary S. Tyson, and Jack W. Davidson. 2006. Exhaustive Optimization Phase Order Space Exploration. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '06)*. IEEE Computer Society, USA, 306–318. doi:10.1109/CGO.2006.15

[23] Prasad A. Kulkarni, David B. Whalley, Gary S. Tyson, and Jack W. Davidson. 2009. Practical exhaustive optimization phase order exploration and evaluation. *ACM Trans. Archit. Code Optim.* 6, 1, Article 1 (April 2009), 36 pages. doi:10.1145/1509864.1509865

[24] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75.

[25] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 216–226. doi:10.1145/2594291.2594334

[26] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. doi:10.1145/1538788.1538814

[27] Zujie Li, Huabiao Qin, and Yixiang Xie. 2025. Applying Knowledge-Guided Deep Reinforcement Learning with Graph Neural Networks for Compiler Optimization. In *Proceedings of the 4th International Conference on Computer, Artificial Intelligence and Control Engineering (CAICE '25)*. Association for Computing Machinery, New York, NY, USA, 193–198. doi:10.1145/3727648.3727682

[28] CClare Liguori. 2025. Introducing Strands Agents, an Open Source AI Agents SDK. https://aws.amazon.com/blogs/opensource/introducing-strands-agents-an-open-source-ai-agents-sdk/.

[29] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 65–79. doi:10.1145/3453483.3454030

[30] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-Refine: Iterative Refinement with Self-Feedback. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.). http://papers.nips.cc/paper_files/paper/2023/hash/91edff07232fb1b55a505a9e9f6c0ff3-Abstract-Conference.html

[31] METR. 2025. Measuring Automated Kernel Engineering. https://metr.org/blog/2025-02-14-measuring-automated-kernel-engineering/.

[32] George C. Necula. 2000. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 83–94. doi:10.1145/349299.349314

[33] Ricardo Nobre, Luiz G. A. Martins, and João M. P. Cardoso. 2016. A graph-based iterative compiler pass selection and phase ordering approach. In *Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems* (Santa Barbara, CA, USA) *(LCTES 2016)*. Association for Computing Machinery, New York, NY, USA, 21–30. doi:10.1145/2907950.2907959

[34] Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. 2025. KernelBench: Can LLMs Write Efficient GPU Kernels? *CoRR* abs/2502.10517 (2025). arXiv:2502.10517 doi:10.48550/ARXIV.2502.10517

[35] Kanghee Park, Jiayu Wang, Taylor Berg-Kirkpatrick, Nadia Polikarpova, and Loris D'Antoni. 2024. Grammar-Aligned Decoding. *CoRR* abs/2405.21047 (2024). arXiv:2405.21047 doi:10.48550/ARXIV.2405.21047

[36] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir R. Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, Joaquin Vanschoren and Sai-Kit Yeung (Eds.). https://datasets-

benchmarks-proceedings.neurips.cc/paper/2021/hash/a5bfc9e07964f8dddeb95fc584cd965d-Abstract-round2.html

[37] Suresh Purini and Lakshya Jain. 2013. Finding good optimization sequences covering program space. *ACM Trans. Archit. Code Optim.* 9, 4, Article 56 (Jan. 2013), 23 pages. doi:10.1145/2400682.2400715

[38] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, dahai li, Zhiyuan Liu, and Maosong Sun. 2024. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=dHng2O0Jjr

[39] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.). http://papers.nips.cc/paper_files/paper/2023/hash/d842425e4bf79ba039352da0f658a906-Abstract-Conference.html

[40] Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob R. Gardner, Yiming Yang, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. 2024. Learning Performance-Improving Code Edits. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. https://openreview.net/forum?id=ix7rLVHXyY

[41] Joar Skalse, Nikolaus H. R. Howe, Dmitrii Krasheninnikov, and David Krueger. 2022. Defining and characterizing reward hacking. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 687, 12 pages.

[42] Richard M. Stallman and GCC DeveloperCommunity. 2009. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. CreateSpace, Scotts Valley, CA.

[43] Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. 2024. LLM4Decompile: Decompiling Binary Code with Large Language Models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 3473–3487. doi:10.18653/v1/2024.emnlp-main.203

[44] Sujun Tang, Christopher Priebe, Rohan Mahapatra, Lianhui Qin, and Hadi Esmaeilzadeh. 2025. REASONING COMPILER: LLM-Guided Optimizations for Efficient Model Serving. arXiv:2506.01374 [cs.LG] https://arxiv.org/abs/2506.01374

[45] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Phoenix, AZ, USA) *(MAPL 2019)*. Association for Computing Machinery, New York, NY, USA, 10–19. doi:10.1145/3315508.3329973

[46] Steven R. Vegdahl. 1982. Phase coupling and constant generation in an optimizing microcode compiler. In *Proceedings of the 15th Annual Workshop on Microprogramming* (Palo Alto, California, USA) *(MICRO 15)*. IEEE Press, 125–133.

[47] Freek Verbeek, Joshua Bockenek, Zhoulai Fu, and Binoy Ravindran. 2022. Formally verified lifting of C-compiled x86-64 binaries. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 934–949. doi:10.1145/3519939.3523702

[48] Siddhant Waghjale, Vishruth Veerendranath, Zhiruo Wang, and Daniel Fried. 2024. ECCO: Can We Improve Model-Generated Code Efficiency Without Sacrificing Functional Correctness?. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, 15362–15376. doi:10.18653/V1/2024.EMNLP-MAIN.859

[49] Anjiang Wei, Tarun Suresh, Huanmi Tan, Yinglun Xu, Gagandeep Singh, Ke Wang, and Alex Aiken. 2025. Improving Assembly Code Performance with Large Language Models via Reinforcement Learning. *CoRR* abs/2505.11480 (2025). arXiv:2505.11480 doi:10.48550/ARXIV.2505.11480