Code Optimization Chain-of-Thought: Structured Understanding and Self-Checking

Qingyao Xu
Academy for Engineering and Technology, Fudan University
China
23210860084@m.fudan.edu.cn

Dingkang Yang
Academy for Engineering and Technology, Fudan University
China
dkyang20@fudan.edu.cn

Lihua Zhang*
Guanghua Lingang Engineering Application and Technology R&D (Shanghai) Co.,Ltd.
China
lihuazhang@fudan.edu.cn

Abstract

In recent years, significant advancements have been made in the field of LLMs (large language models), particularly within the domain of code optimization. This paper explores the realm of code optimization in LLMs and presents comprehensive approaches to enhance the model's abilities to generate and correct code through fine-tuning, training, and and applying Chain-of-Thought techniques during the inference phase. Novel strategies are introduced to augment the model's understanding of coded structures during the fine-tuning phase by integrating structured code information, providing a more robust grasp of core principles. This knowledge augmentation reflects a significant improvement in the model's structured comprehension of code and lays the foundations for a more effective generation and revision of code. Furthermore, a unique Chain-of-thought technique is applied during the inference phase to generate core coding principles and several sets of unit test data. The large language model is empowered to utilize these testing datasets for an active self-check and modification process. This novel methodology fosters the model's ability to autonomously adjust and fix the produced code, thereby enhancing the overall robustness and reliability of the generated code. The concepts and techniques elucidated in this paper aim to carve a path for future research and advancements in large language model code optimization.

CCS Concepts

• **Computing methodologies** → Artificial intelligence; Natural language processing; Natural language generation.

Keywords

Artificial Intelligence, Large Language Models, Code Optimization, Chain of Thought, Fine-tuning

*Corresponding author.

CAIBDA 2024, June 21–23, 2024, Zhengzhou, China

ACM ISBN 979-8-4007-1024-7/24/06
https://doi.org/10.1145/3690407.3690479

ACM Reference Format:
Qingyao Xu, Dingkang Yang, and Lihua Zhang. 2024. Code Optimization Chain-of-Thought: Structured Understanding and Self-Checking. In *2024 4th International Conference on Artificial Intelligence, Big Data and Algorithms (CAIBDA 2024), June 21–23, 2024, Zhengzhou, China.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3690407.3690479

1 Introduction

Although the work of optimizing compilers and other program optimizations is impressive [1], traditional methods are often based on manual analysis and tuning, requiring specialized knowledge and experienced programmers to complete. In recent years, substantial work has demonstrated the unique advantages of machine learning for performance optimization [2]. However, these techniques are either restricted in scope or hard to develop further because of the absence of open datasets and the insufficiency of reliable performance measurement techniques, which has hindered the research in this aspect. Fortunately, a new benchmark for performance optimization has been introduced that addresses the critical challenges of reproducible performance measurement and provides an extensive evaluation of the various adaptive techniques based on it [3]. Existing code optimization during the inference stage is often a one-time process, lacking iterative revision and improvement. Even if LLMs generate code that appears correct, they cannot verify its logical and syntactic validity independently. Large models are primarily trained on text data to generate text or answer natural language tasks, not to understand code structure and semantics. To address this, we propose a methodology that uses code tokens, linearized Abstract Syntax Tree (AST), and natural language inputs to enhance the model's code comprehension. During inference, the model can iteratively generate and refine code by formulating test cases and self-examining.

First, we construct a Simplified AST Performance-Improving Edits (SAPIE) dataset. SAPIE is built on the Performance-Improving Edits (PIE) dataset, which collects C++ programs written to solve competitive programming problems, where we track a single programmer's submissions as they evolve, filtering for sequences of edits corresponding to performance improvements [3]. The key is building the simplified AST sequence for each piece of code and adding it to the dataset. Next, we fine-tune multiple models and test their performance. We evaluate performance using the gem5 CPU simulator, the gold standard CPU simulator in academia and industry, and model the most advanced general-purpose processors [4]. This evaluation strategy is entirely deterministic, ensuring reliability and repeatability. Finally, we propose an advanced inference

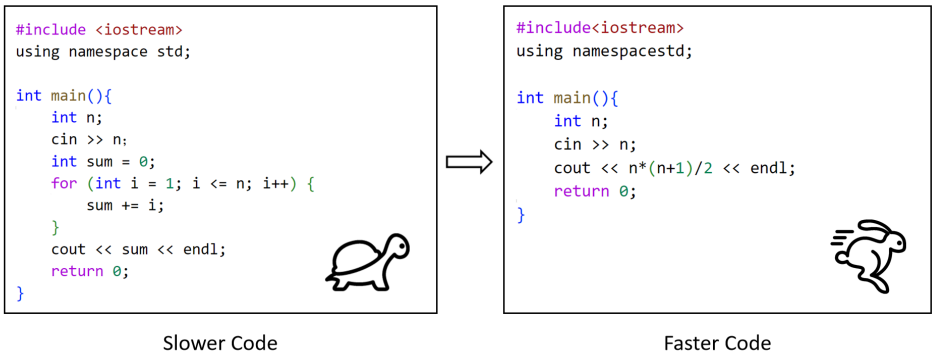

Figure 1: Example of a program to solve the problem of "Calculating the sum of integers 1 to N". The program on the left runs in O (N), while the program on the right runs in O (1). There are 77 K C++ program pairs in the SAPIE dataset.

strategy: iterative inference and self-correction. The invention method is mainly divided into Base Code-CoT and Self-Critical Code-Cot. The Base Code-CoT method starts with sample inputs and a Chain of Thought for code optimization. The Self-Critical Code-CoT method adds a self-check layer, where the LLM designs test cases, and iterates until the code is logically and syntactically correct with improved performance.

In summary, our contributions are:

- We have built a new code dataset containing 77k C++ program pairs and their improved AST sequences that can be used to improve the structured understanding of LLMs.
- Based on the built data set, we fine-tune multiple models. Experiments show that our dataset can improve the performance of LLMs in code generation.
- In the inference phase, we propose a strategy of Chain-of-Thought, which enables the model to iteratively generate code and improve its output code to increase the reliability and robustness of the code by generating principles, formulating test cases, and combining self-inspection. Overall, our best model, CodeQwen1.5-13B-Chat achieves an average speedup of 7.53×, and the correct rate of generated code increases to 96.54%.

2 Simplified AST Performance-Improving Edits (SAPIE) Dataset

We constructed a dataset to facilitate the adaptation of code LLMs for performance optimization, focusing on improving execution time and code structure. Built on performance-improving edits (PIE) from CodeNet [5], we focus on C++ programs due to their performance-oriented nature and gem5 simulator compatibility [4].

When dealing with the programs written by users for a specific problem, we follow a series of processes. First, we sort by CPU execution time based on the code for a particular task. Let Y_k={y_1,y_2,y_3... } be a sequence of code in chronological order, with k representing a task. We remove those not accepted by the automated system, filtering out incorrect programs or those exceeding the allowed time. Then, we construct pairs P = (y_1, y_2),(y_2, y_3),(y_3, y_4). . .. . .from the resulting trajectory of programs, only keeping the pairs where the relative time improvement is more than 10% ($\frac{(\text{time}(y_i)-\text{time}(y_k))}{time(y_i)} > 10\%$), Figure 1 is an example of a code pair.

Table 1: details of the SAPIE dataset.

Dataset	Number of Unique Problem IDs	Number of pairs
Train	1,474	77,967
Val	77	2,544
Test	41	978

Statistical analyses on the SAPIE dataset. We manually analyze 120 randomly sampled (source, optimized) program pairs to identify the algorithmic and structural changes responsible for these improvements. The transformations fall into four main categories: ①Algorithmic changes, such as replacing recursive methods with dynamic programming or simplifying constructs by omitting Binary Indexed Trees, are the most common, making up 34.15% of the changes. ②Input/output operations, like switching from 'cin/cout' to 'scanf/printf' or optimizing string reading, account for 26.02%. ③Data structure modifications, including replacing vectors with arrays, constitute 21.14%. ④Miscellaneous adjustments, such as code cleanups and constant optimizations, comprise 18.70%.

The dataset is split into train/validation/test sets, ensuring no specific competitive programming problem appears in multiple sets. The dataset details are shown in Table 1.

Structured information. Structural information is crucial for understanding source code. AST is extensively utilized in code-related tasks (e.g., [6] [7]) to represent the structural information of code, encompassing rich syntactic structural details that code sequences cannot convey. However, we did not directly use the preorder traversal AST or AST with SBT traversal as our input, even though these have often proved useful [8]. Because the simple linearized AST is long, it is usually more than three times the length of the original code. To solve this problem, we propose a simplified version of AST called SAST. SAST can reduce the length of the generated traversal sequence by more than half, and Figure 2 shows a comparison.

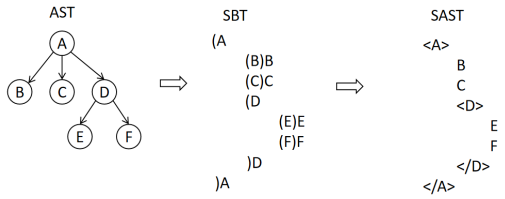

Figure 2: A comparison example of SBT and SAST. We found that the length of SAST was reduced significantly compared to SBT. Indentation is used to better understand the sequence without line breaks.

SBT starts with "(" and the node name, ends with ")" and the node name. In SAST, we do not need start and end characters for the terminal nodes, and we use an XML-like markup strategy for the start and end of non-terminal nodes. Specifically, we use the same start character as SBT and replace the end character with "/", node name, and ")". This can minimize the sequence length without confusing the terminator with the terminal node.

Test cases. Test cases play a crucial role in our dataset. We evaluate the correctness through unit tests and reject programs with a single failed test. We incorporate additional test cases from AlphaCode [9] generated by a fine-tuned LLM to enhance coverage. The number of test cases per problem varies in different sets.

3 Adapt code LLMS to program optimization

3.1 Fine-tuning

In this section, we will use a SAPIE dataset fine-tune the LLMs, demonstrating that fine-tuning with such dataset can help the model better understand logical relationships and thus better perform code optimization tasks.

The model input consists of three parts: (1) raw code (src_code), (2) Structured Code Representation (SAST), and (3) a natural language prompt describing the task. Input and output are shown in Figure 3. By combining the original code, SAST, and natural language descriptions, the model is able to better understand the logic of the code and the optimization objectives, thus performing more accurately and efficiently when generating optimized code. In Section 4, based on the experimental results, we verify the effectiveness of fine-tuning with data sets with structured information for code optimization tasks.

3.2 Self-Checking Code Chain-of-Thought

Inspired by Chain-of-Thought technology [10], we propose an inference strategy that allows models to iteratively verify and correct their outputs to ensure the correctness of optimization and functionality. This strategy is called the Self-Checking Code Chain of thought. It is divided into two stages: Base Code-CoT and Self-Critical Code-CoT. The specific process is shown in Figure 4.

Base Code-CoT has two main tasks. 1. Use LLMs to decompose tasks, provide a logical "thinking chain" for optimizing target code tasks, and explain in detail how to process tasks so as to generate optimized code. 2. Subsequently, regenerated into test cases, LLM developed a set of test cases for the subsequent measurement of the accuracy and effectiveness of the generated code.

Self-Critical Code-CoT introduces a self-test layer to test Output Optimized Version Code using the test cases generated in the previous stage. If the Code has errors or fails the test case, etc., the Testing result of the code is obtained.

Test Results mainly include four kinds of results: ① **Syntax error**: An exception thrown because undefined or incorrect syntax is used. Syntax errors are detected during compilation. ② **Run timeout**: The running of a program that does not end within a specified period of time usually occurs during a dead loop. ③ **Not pass**: Failed to pass the test cases. ④ **Unoptimization**: CPU execution time did not increase by more than 10%.

The error report and the current code are re-entered as input, allowing the LLMs to revise the code based on the new input, ensure that the code is competent and aligned with the initial prompt, and execute correctly. The iterative process of testing and self-correcting will be carried out through a series of multi-step iterations, allowing for a user-specified maximum iterations. This iteration ends when the LLM produces code that is also logically and syntactically correct with improved performance.

4 Evaluation

Models. To achieve the best results, we chose CodeLlama and CodeQwen among the many types of large models available. CodeLlama was selected due to its strong baseline performance in code generation and its architectural efficiency, which makes it suitable for fine-tuning tasks. [11] On the other hand, CodeQwen was chosen for its advanced capabilities in handling complex programming logic and its robust performance in prior studies involving code optimization. [12] These models provide a balanced foundation for further enhancements using our proposed methodology. And our results suggest that pre-trained code LLMs have limitations in their capacity to optimize code in the absence of a dataset such as SAPIE. We evaluate and adapt models from the CodeLlama models [11] and CodeQwen1.5 models [12]. We used pre-trained checkpoints of

```
This is a slow program we want to optimize it.
### Program:{src code}
###Structured representation of code:{SAST}
###Optimized Version:
```

```
###Optimized Version:
{fast code}
```

Figure 3: Training prompt (left) and target (right) for Goal-Conditioned optimization with SAPIE. "{src_code}" is the original code to be optimized. "{SAST}" is a structured representation of the code. "{fast_code}" is the optimized version of the code.

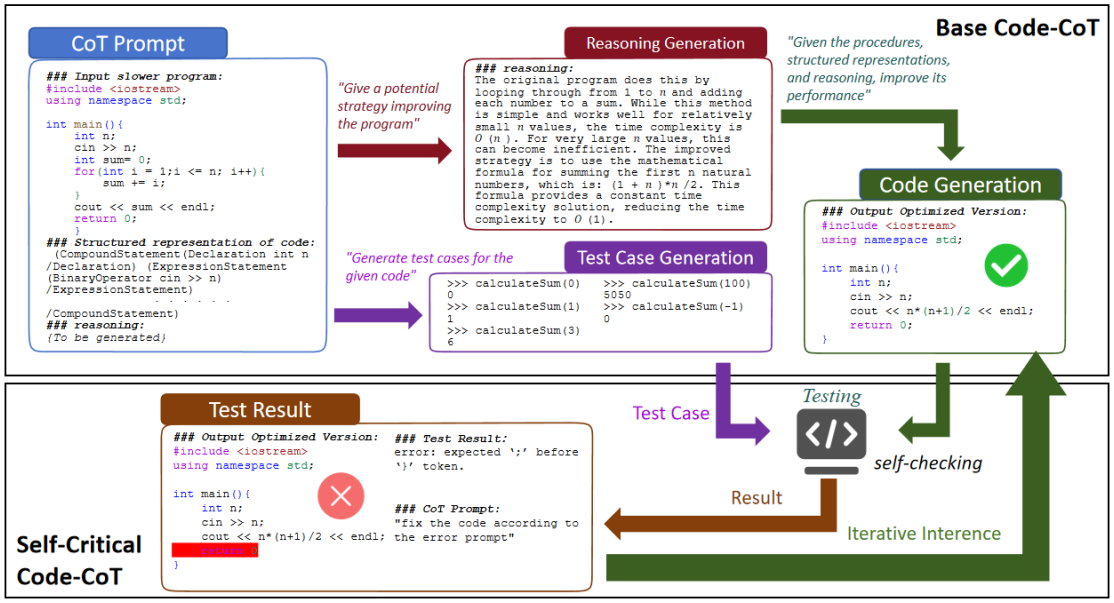

Figure 4: Flow chart of Self-Checking Code Chain of thought. It contains two stages, i.e., Base Code-CoT and Self-Critical Code-CoT.

Table 2: This table reports the evaluation results of multiple scenarios. We include the experiment configuration to the right of the model name. The highest number in each column is bolded, and the second-highest is underscored.

Scenario	Models	%Opt	Speedup	%Correct
Human Reference	-	88.42%	3.66×	95.67%
Prompt	CodeLlama 13B	19.63%	1.30×	78.73%
	CodeQwen1.5 13B	29.05%	1.62×	88.72%
Fine-tuning	CodeLlama 13B	68.56%	5.65×	70.96%
	CodeQwen1.5 13B	86.54%	6.86×	92.41%
Self-Checking Code-CoT	CodeLlama 13B	55.60%	3.5×	94.61%
	CodeQwen1.5 13B	68.76%	3.88×	92.55%
Fine-tuning +	CodeLlama 13B	83.84%	6.89×	89.75%
	CodeQwen1.5 13B	**89.56%**	**7.53×**	**96.54%**

CodeLlama 13B obtained via HuggingFace [13]. For the CodeLlama family of models, we use the base set of models that have not been instruction-tuned, as the authors of the paper note that instruction-tuning diminished the performance of code generation. The model fine-tuned by the SAPIE dataset performed significantly better on the code optimization task, especially when combined with the Self-Checking Code Chain of Thought technology; the %Opt and %Correct of the model were significantly improved; specific results are shown in Table 2. We used a batch size of 32 and a learning rate of 1e-5 for all of the experiments.

Metric. To evaluate performance, we measure the following for functionally correct programs:

- **Percent Optimized** [%Opt]: This metric indicates how many outputs the CPU executes more than 10% faster than the original code. We assume that there are n outputs that satisfy our optimization requirements and that the test set contains 978 data. Then, we can calculate the percent optimized.

$$\%\text{Opt}\,(n) \;=\; \left(\frac{n}{978}\right) \times 100\%$$

- **Speedup**: The rate at which code execution time is increased. If "s", "f" means "slower code" and "faster code", the faster code must be correct. Then, we will show how to compute the Speedup below.

$$\text{Speedup}\;(s,\,f) \;=\; \left(\frac{s}{f}\right)$$

- **Percent Correct** [%Correct]: The output code executes correctly and passes all test cases (be functionally consistent with the original code).

By providing structured information and performance improvement edits, the SAPIE datasets help models better understand code logic and structure to perform better in optimization tasks. This

Table 3: Error analysis of CodeQwen1.5 fine-tuned with SAPIE dataset.

ScenarioResult	Percentage
Failed to compile (syntax/type errors)	12.57%
Compiled, but got [95-100%] of test cases wrong	27.88%
Compiled, but got (75, 95%] of test cases wrong	12.09%
Compiled, but got (25, 75%] of test cases wrong	10.27%
Compiled, but got (0-25%] of test cases wrong (at least 1 test case was wrong)	12.47%
Ran all test cases, the program was not 1.1× speedup or higher	24.72%

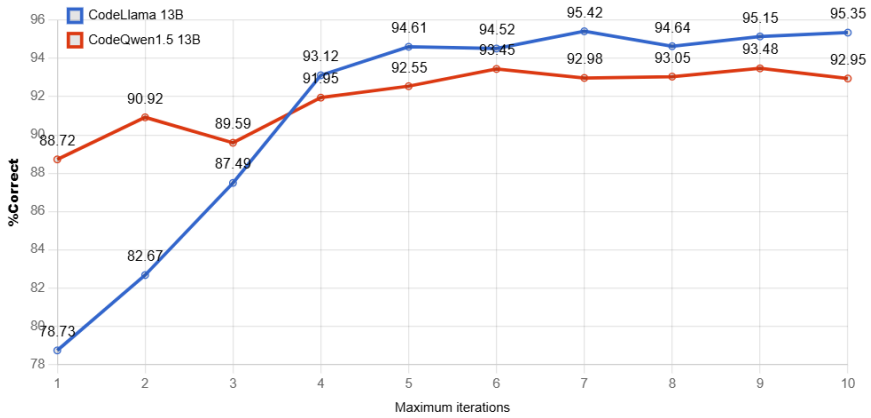

Figure 5: Results when Maximum iterations go from 1 to 10.

shows that building high-quality training data sets is crucial to improving the performance of large language models.

In addition, we found that the %Correct after fine-tuning decreased, which may be different from the tasks, data types or distributions of SAPIE and the original training data set, and the performance labels may restrict the set of generated programs that are correct. We analyzed the causes of errors, as shown in Table 3.

In previous experiments, we set the default maximum iterations for Self-Checking to 5. To understand its impact on code correctness (%Correct), we adjusted the maximum Self-Checking iterations from 1 to 10, keeping other variables constant. Figure 5 shows that code correctness improves with more iterations but plateaus or decreases beyond a certain threshold due to new errors introduced by excessive iterations.

The experimental results show that fine-tuning using structured information and natural language prompts can significantly improve the performance of large models in code optimization tasks. Especially with the combination of our Self-Checking Code-CoT, the model can not only generate optimized code but also self-check and correct it, improving the reliability and robustness of code generation. In practical applications, it is recommended to set the maximum iterations for Self-Checking to a reasonable range (such as 5) to balance the improvement in correctness and the risk of introducing errors.

5 Conclusion

This study explored large language models (LLMs) for code optimization and correction by fine-tuning with our SAPIE dataset and incorporating chain-of-thought (CoT) techniques. Fine-tuning provided structured information and performance-improving edits, significantly enhancing the models' ability to understand and optimize code. The integration of self-checking CoT allowed models to iteratively verify and correct their outputs, ensuring both optimization and functional correctness. Our results demonstrated that fine-tuned models combined with self-checking CoT techniques achieved substantial improvements, with CodeLlama 13B and CodeQwen1.5 13B reaching optimization rates of 83.84% and 89.56%, speedup factors of 6.89× and 7.53×, and accuracy rates of 89.75% and 96.54%, respectively. These findings highlight the effectiveness of high-quality datasets and advanced processing techniques in enhancing LLM performance for code optimization tasks.

Acknowledgments

This research was funded by Shanghai 2021 "Science and Technology Innovation Action Plan" (Project No. 21XD1430300).

References

[1] Aho, A. V., Sethi, R., & Ullman, J. D. 2007. Compilers: Principles, Techniques, and Tools (Vol. 2). Addison-Wesley Reading.

[2] Mankowitz, D. J., Michi, A., Zhernov, A., Gelmi, M., Selvi, M., Paduraru, C., ... & Silver, D. 2023. Faster sorting algorithms discovered using deep reinforcement learning. Nature, 618(7964), 257-263.

[3] Shypula, A., Madaan, A., Zeng, Y., Alon, U., Gardner, J., Hashemi, M., ... & Yazdanbakhsh, A. 2023. Learning performance-improving code edits. arxiv preprint arxiv:2302.07867.

[4] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., ... & Wood, D. A. 2011. The gem5 simulator. ACM SIGARCH computer architecture news, 39(2), 1-7.

[5] Puri, R., Kung, D. S., Janssen, G., Zhang, W., Domeniconi, G., Zolotov, V., ... & Reiss, F. 2021. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. arxiv preprint arxiv:2105.12655.

[6] Wang, Y., & Li, H. 2021, May. Code completion by modeling flattened abstract syntax trees as graphs. In Proceedings of the AAAI conference on artificial intelligence (Vol. 35, No. 16, pp. 14015-14023).

[7] Zhang, J., Wang, X., Zhang, H., Sun, H., & Liu, X. 2020, June. Retrieval-based neural source code summarization. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (pp. 1385-1397).

[8] Hu, X., Li, G., **a, X., Lo, D., & **, Z. 2018, May. Deep code comment generation. In Proceedings of the 26th conference on program comprehension (pp. 200-210).

[9] Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., ... & Vinyals, O. 2022. Competition-level code generation with alphacode. Science, 378(6624), 1092-1097.

[10] Wei, J., Wang, X., Schuurmans, D., Bosma, M., **a, F., Chi, E., ... & Zhou, D. 2022. Chain-of-thought prompting elicits reasoning in large language models. Advances in neural information processing systems, 35, 24824-24837.

[11] Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., ... & Synnaeve, G. 2023. Code llama: Open foundation models for code. arxiv preprint arxiv:2308.12950.

[12] Bai, J., Bai, S., Chu, Y., Cui, Z., Dang, K., Deng, X., ... & Zhu, T. 2023. Qwen technical report. arxiv preprint arxiv:2309.16609.

[13] Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., ... & Rush, A. M. 2020, October. Transformers: State-of-the-art natural language processing. In Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations (pp. 38-45).