Parallel Code Generation with Large Language Model

Xiaowen Huang*
Shenzhen University
Shenzhen, China

Xu Zhang
Southern University of Science and Technology
Shenzhen, China

Lvfang Tao, Renjie Mao, Wenxi Zhu, Mingwen Deng
Tencent
Shenzhen, China

Nan Zhou
Shenzhen Polytechnic University
Shenzhen, China

Jintao Meng, Yanjie Wei
Shenzhen Institutes of advanced Technology, CAS
Shenzhen, China

Amelie Chi Zhou
Hong Kong Baptist University
Hongkong, China

Bingqiang Wang
Peng Cheng Laboratory
Shenzhen, China

Shengzhong Feng
Guangdong Institute of Intelligence Science and Technology
Zhuhai, China

Abstract

High-performance parallel code generation is a complex and fascinating area in computer science that focuses on producing code that executes as quickly and efficiently as possible. In our paper, we designed a new architecture for parallel code generation agent with 4 inter-connected components of *LLM—Memory, Planning, Tools and Action.* It also incooperated with two techniques: data augmentation, prompting and retrieval-augmented editing to improve the performance of the parallel codes. Data augmentation is implemented by extracting and processing PIE dataset, and also synthesis dataset generated by LLM models with ParEval benchmark. Finally planning-oriented prompting, code verification and retrieval augmented editing are used to promote the actual performance of the LLM generated code. The evaluation results confirm that a rough speedup of 6.06X and 5.13X are achieved using Qwen2.5-Coder-7B-Instruct, Qwen2.5-Coder-14B-Instruct LLM models.

Keywords: LLM, Code Generation, Parallelization

1 Introduction

High-performance parallel code generation is a complex and fascinating area in computer science that focuses on producing code that executes as quickly and efficiently as possible. It involves numerous techniques, algorithms, data structures, parallel performance-oriented programming, and considerations at different memory hierarchy on various levels, from the source code itself down to the specific hardware architecture. Large language model (LLM) based coding and optimization researches can be classified into three categories: benchmark dataset, code generation, and code optimization. Popular open bannchmark datasets are ParEval [17], HumanEval [2], PIE [25] and CodeNet [21]. StarCoder [11], AlphaCode [12], WizardCoder [16], Magicoder [28], CodeLlama [22], MetaGPT [7], and DeepSeek-Coder [6] are state-of-the-art tools for code generation; HPC-Coder [18], PloyCoder [30], and PIE [25] are three example tools used for code optimization.

Most of the above code generation researches are promising in improving the productivity of developers and the overall quality of software. However, the performance of the generated code is critical with rapid advancement in computing chips and LLMs in recent years. Generating high-performance parallel code is a particularly complex task; it involves reasoning about data distributions, parallel algorithms, and parallel programming models. In this work, we explore the techniques that can boost the performance of LLM generated parallel codes. In our paper, we designed a new architecture for Parallel Code Generation Agent, incooperated with two techniques: data augmentation, prompting and retrieval-augmented editing to improve the performance of the parallel codes. Our contributions are four-fold:

❶ **A new agent architecture** for Parallel Code Generation is proposed with 4 components inter-connected to the LLM —*Memory*, *Planning*, *Tools* and *Action*. Each components connected to the main agent, and is designed to address key aspects of the optimization pipeline.

❷ **Data augmentation** is applied to generate two datasets, one is the internal dataset with 72,126 entries extracted and processed from the PIE dataset, the other one is the synthesis dataset with 30,000 entries, is generated by inferencing the ParEval dataset using LLM API calls or open source LLM models and filtered with our Action components.

❸ **Prompting and retrieval augmented** is applied with planning-oriented prompting, code verification and retrieval augmented editing to promote the actual performance of the LLM generated codes. The three methods are not used stand-alone, they are routed and cooperated between each

*Xiaowen Huang, Xu Zhang and Lvfang Tao contributes equally. The corresponding author is Jintao Meng (jt.meng@siat.ac.cn).

FCPC'25, March 01–05, 2025, Las Vegas, California, USA
2025.

other to filter low quality results and prompt to generated high quality codes.

❹ **Positive improvement** is confirmed in our experiment with three LLM models (GPT-4o-mini, Qwen2.5-Coder-7B-Instruct, Qwen2.5-Coder-14B-Instruct). The evaluation results confirm that a rough speedup of 6.06X and 5.13X are achieved by using Qwen2.5-Coder-7B-Instruct, Qwen2.5-Coder-14B-Instruct LLM models.

The rest is organized as follows: Section 2 introduces the three major techniques used to boost the performance of LLM generated parallel codes. The evaluated results and performance discussion are presented in Section 3. Section 4 provides a brief discussion to the related work. Finally, Section 5 concludes this work.

2 Methods

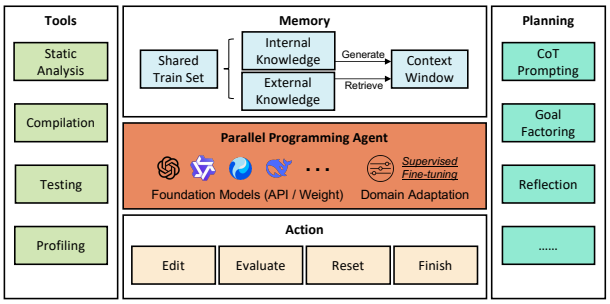

Figure 1. Architecture of **Parallel Code Generation**.

2.1 Architecture of Parallel Code Generation Agent

In light of the recent successes of autonomous LLM agents in addressing complex problem solving tasks through self-directed planning, action, and reasoning [27], we propose a novel Architecture for **Parallel Code Generation**, as shown in **Figure 1**. The architecture comprises four components interconnected to the LLM —*Memory*, *Planning*, *Tools* and *Action* — each connected to the main agent, and is designed to address key aspects of the optimization pipeline:

1. **Memory**: The memory system unifies both internal knowledge representations and external knowledge in model weights. The internal knowledge representations are learned by fine-tuning LLM over the internal data set described in Section 3.2.1, and the external knowledge representations are retrieved via retrieval-augmented generation (RAG) over the external data set described in Section 3.2.2. Both datasets have performance-annotated samples and can inject domain-specific knowledge to the agent.
2. **Planning**: Planning is pivotal to the prompting workflow, including both single-stage prompts for direct parallelization and decomposed workflows for multi-stage code transformation. These methods, which improve the alignment of generated code with optimization goals, are discussed in Section 2.3.1.
3. **Tools**: This module handles static analysis and compiling-based verification, critical for validating code correctness and compliance with competition rules. The details are covered in Section 2.3.2. Additionally, we consider testing and profiling capabilities for potential iterative workflows tailored to fit real-world performance tuning scenarios.
4. **Action**: The action space defines the agent's capabilities for state transitions during the code optimization process. Possible state transitions include edits, evaluations, and resets, which can be autonomously controlled by the LLM or guided by rule-based heuristics.

This modular design enables the agent to effectively integrate planning, retrieval, and execution, addressing the challenges of efficient parallel programming with a flexible and adaptive approach.

2.2 Data Augmentation

2.2.1 Public Datasets. The publicly available PIE dataset [25] is selected and post-processed by deleting long samples as our **Internal Dataset**. PIE dataset originally contains 77,967 entries of serial code optimization data. Each entry in the PIE dataset[25] includes the code before optimization, the code after optimization, the execution times of both code versions, the speedup ratio, and the normalized integer speedup ratio. Due to the presence of excessively long data in the PIE dataset, which may result in input sequences exceeding the length limit for large language models (LLMs). Thus, we tokenized the data using the o200k_base tokenizer model employed by GPT-4 [20]. All long entries with token counts exceeding 1,000 will be eliminated, and finally 5,841 of these long entries were removed from the PIE dataset.

To ensure consistency with the speedup ratio data in Section 3.1.2, we normalized the speedup ratios to floating-point numbers between 1 and 10, allowing for a clearer comparison of code quality for the same problem across different implementations. After these post-processing, the final data set keeps 72,126 entries.

2.2.2 Data Synthesis via Self-Play. Considering the lack of performance-annotated data on parallel programming in publicly available code datasets [19, 25], we utilized different open-source models and APIs to synthesize optimized code for data augmentation. We perform multiple inferences on 60 questions from the ParEval dataset [17], collecting 500 effective samples for each question, and in total 30,000 distinct target code examples are gathered. We use the methods introduced in 2.3.1 for code generation.

For API calls, we experimented with `gpt-4o-mini` and `deepseek-chat` (powered by DeepSeek-V3). For open source models, we employ vLLM for batch-accelerated generation. Using regular expressions, we extracted the target code from the model output and recorded the compilation information, along with the speedup information relative to the source

code of the questions. We normalized the speedup of different target codes for each question, then ranked all target codes based on the normalized speedup, and extracted the higher-value portions as training data.

For the extracted code, we perform rigorous verifications to ensure code correctness and exclude inaccurate speedup measures. Initially, We discovered some outliers in the speedup records of the synthesized data and identified two reasons for this: (1) erroneous code that occasionally passed all the program-generated test cases; (2) the speedup of some code is highly dependent on the test data. We improved the testing method by conducting multiple tests on the same target code using random test cases. If any test fails, the code is considered unusable. If all tests pass, we calculate the average speedup after removing extremums to mitigate noisy speedup signals.

The synthesized data serves dual purposes in our agent architecture: (1) It expands the **Memory** component's external knowledge base through performance-annotated examples for RAG retrieval, (2) Provides performance-annotated training sample for internal knowledge acquisition through fine-tuning.

2.3 Prompting and Retrieval Augmented

2.3.1 Planning-Oriented Prompting. Our **Planning** component covers various optimization strategies through prompt engineering. The four prompt types represent different planning granularities:

1. *parallel*: Directly transform the serial source code into efficient OpenMP parallel code.

2. *serial*: Do not explicitly specify a parallelization method; instead, instruct the model to rewrite and optimize the serial source code.

3. *any*: Suggest OpenMP parallel optimization as an available option, requiring the model to rewrite the serial source code into faster code without restrictions on serial or parallel execution.

4. *multi-stage*: Decompose the optimization process into two stages: first, perform `serial` optimization, and then apply `parallel` to convert the code optimized in the first stage into parallel code.

2.3.2 Code Verification. The verification pipeline implements core functionality of the **Tools** component through a 5-step verification to the LLM-generated parallel code: ❶ The C++ code are extracted from the output json file generated by LLM. ❷ We compile the generated C++ code into object files using GCC to verify its compliance with the C++ and OpenMP syntax specifications. ❸ To address potential inconsistencies in function interfaces that may arise from large language models (LLMs), we employ the nm tool to analyze the abstract syntax tree (AST) of the compiled object files. This enables a systematic comparison of function interface consistency between the compiled object files and those derived from the original source code. ❹ When inconsistencies or syntax errors are detected, a differentiated approach is applied based on the characteristics of the model. ❺ For LLMs with larger parameter sizes or broader context windows, the error information is explicitly fed back to the model, prompting it to regenerate the code. For models with smaller parameter sizes or limited context windows, immediate code regeneration is initiated without additional feedback. The above five-steps' iterative process continues until either syntactically correct code is successfully generated or the predefined maximum iteration threshold is reached, thereby ensuring the reliability and robustness of the generated code.

A **self-correction mechanism** is applied by using prompt engineering. As the code generated by LLMs may potentially yield incorrect results, we implement a self-correction mechanism that enables the LLM to predict, verify, and rectify the functionality of the generated code. Specifically, we instruct the LLM to assess whether the generated code maintains functional equivalence with the source code, requiring it to provide detailed explanations and concrete examples to validate its evaluation. Based on this analysis, the LLM then regenerates the code to ensure functional correctness and the correctness rate is further improved with this technique.

2.3.3 Retrieval Augmented Editing. The retrieval-augmented generation (RAG) [10] approach can be used to enhance the capabilities of language models (LLMs) on the complex and knowledge-intensive task of program optimization. RAG is used for program optimization demands LLM's deep expertise in algorithms, data structures, and parallel performance-oriented programming. By retrieving highly relevant examples, RAG can significantly improving LLM's ability to generate high efficient and parallelized solutions.

Our RAG implementation directly operationalizes the **Memory** component through combining both Internal and Synthetic dataset as the RAG's knowledge database. The source code of the problems in both datasets and the targeting problem needs to be optimized are encoded with a BERT model. This step transforms the textual information into numerical vector representations, known as embeddings. The resulting embeddings are then indexed using FAISS [8], enabling efficient similarity-based retrieval. When a new program is received for optimization, the system retrieves the L most similar programs from this knowledge base based on their embeddings, and for practical cases, L can be set to be $1, 2, 3$.

Each retrieved example includes both the original unoptimized version and its optimized counterpart. These paired examples serve as the foundation for dynamically generating prompts that are specifically tailored to the source code of the problem at hand. The custom prompts provide contextual information and optimization cues, guiding the LLM more accurately in understanding and processing the current task. Finally, these tailored prompts are processed by the LLM to

Table 1. Performance Metrics: Speedups for different workflows.

Type	Model	Avg Speedup across K runs			Aggregated Best Speedup across K runs		
		$K = 1$	$K = 10$	$K = 100$	$K = 1$	$K = 10$	$K = 100$
Direct	Qwen2.5-Coder-7B-Instruct	1.81	1.87	**1.86**	1.81	2.32	2.87
	Qwen2.5-Coder-14B-Instruct	2.31	2.17	2.16	2.31	2.80	3.05
Serial	Qwen2.5-Coder-7B-Instruct	1.35	1.36	1.32	1.35	1.80	1.96
	Qwen2.5-Coder-14B-Instruct	1.50	1.44	1.43	1.50	1.60	1.71
Parallel	Qwen2.5-Coder-7B-Instruct	1.36	1.35	1.38	1.36	1.86	2.40
	Qwen2.5-Coder-14B-Instruct	1.50	1.50	1.55	1.50	1.72	2.09
Multi-Stage	Qwen2.5-Coder-7B-Instruct	**1.86**	1.65	1.69	1.86	2.45	**2.97**
	Qwen2.5-Coder-14B-Instruct	2.28	2.23	2.29	2.28	2.76	**3.20**
RAG	Qwen2.5-Coder-7B-Instruct	1.66	1.79	1.79	1.66	2.22	2.68
	Qwen2.5-Coder-14B-Instruct	1.95	1.97	2.04	1.95	2.32	2.77
RAG + Self-Match	Qwen2.5-Coder-7B-Instruct	3.29	3.77	**3.81**	3.29	5.46	**6.06**
	Qwen2.5-Coder-14B-Instruct	3.98	3.94	**3.75**	3.98	4.84	**5.13**

perform further optimization, with a focus on enhancing the parallelism and performance of the generated code.

3 Experimental Results

The speedup numbers achieved by different models are presented in Table 1 for the optimization of the source code in various workflows. The term *Avg Speedup across @K runs* refers to the average speedup obtained from generating the target code *K* times, while *Aggregated Best Speedup across @K runs* denotes the maximum speedup achieved from generating the target code *K* times. The workflows corresponding to *Parallel*, *Serial*, *any*, and *multi-stage* are detailed in 2.3.1. The workflows *Rag* and *Rag+Self-Match* are described in 2.3.3. We observe that for different prompts, directly instructing the model to generate OpenMP parallel code (*parallel*) improves the quality of the generated code compared to not enforcing parallel code generation (*serial*, *any*). For the approach of first optimizing the code serially and then converting it to OpenMP parallel code (*multi-stage*), the average speedup across multiple generations does not show significant improvement, but the maximum speedup does increase. With the introduction of retrieval-augmented generation (RAG), the prompt includes the two most similar {source code: target code} pairs to guide the model. The default version of RAG can retrieve two samples with identical source code, which, as observed from the experimental results, can negatively impact the quality of the generated code. Therefore, we incorporated self-match verification to ensure that the two retrieved samples have different source codes, significantly enhancing the code quality.

4 Related work

The ultimate goal of parallel codes is to achieve theoretical-peak performance of the target hardware or chips by implementing the lowest complexity algorithm for any given problem. The following three stages are needed to achieve the best theoretical peak performance:

Node-level Performance Engineering is an independent research direction named high performance computing. All of the algorithms in ParEval[1, 3–5, 13, 14, 17, 23, 26, 29] are manually developed by highly skilled experts in high-performance computing. This method can achieve the best performance but at the cost of huge human resources.

Compiler-level optimization rely on compiler optimization. It involves loop unrolling, cache optimization, instruction scheduling, vectorization, etc. [9, 15, 24]. There still has to be a performance gain compared with the first stage.

LLM-Powered Code Parallelization LLM can potentially help developers overcome these challenges. The advantage of this method is that it is fully automatic, but the ultimate performance of the generated code is still valuable to evaluate with these codes developed by experts.

5 Conclusion

High-performance parallel code generation involves numerous techniques, algorithms, data structures, parallel performance-oriented programming, and considerations at different memory hierarchy on various levels, from the source code itself down to the specific hardware architecture. In this work, we designed a new architecture for Parallel Code Generation Agent, in-cooperated with two techniques: data augmentation, prompting and retrieval-augmented editing to improve the performance of the parallel codes. A speedup of 6.06X and 5.13X are achieved by using Qwen2.5-Coder-7B-Instruct, Qwen2.5-Coder-14B-Instruct LLM models on ParEval dataset.

Acknowledgments

This work is supported by the Shenzhen-HongKong Joint Funding Project (Category A) under Grant No. SGDX20230116092056010, Shenzhen Key Laboratory of Intelligent Bioinformatics under Grant No. ZDSYS20220422103800001.

References

[1] Junya Arai, Masahiro Nakao, Yuto Inoue, Kanto Teranishi, Koji Ueno, Keiichiro Yamamura, Mitsuhisa Sato, and Katsuki Fujisawa. 2024. Doubling Graph Traversal Efficiency to 198 TeraTEPS on the Supercomputer Fugaku. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Atlanta, GA, USA) *(SC '24)*. IEEE Press, Article 101, 14 pages. https://doi.org/10.1109/SC41406.2024.00107

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] https://arxiv.org/abs/2107.03374

[3] Tun Chen, Haipeng Jia, Yunquan Zhang, Kun Li, Zhihao Li, Xiang Zhao, Jianyu Yao, and Chendi Li. 2023. OpenFFT: An Adaptive Tuning Framework for 3D FFT on ARM Multicore CPUs. In *Proceedings of the 37th ACM International Conference on Supercomputing* (Orlando, FL, USA) *(ICS '23)*. Association for Computing Machinery, New York, NY, USA, 398–409. https://doi.org/10.1145/3577193.3593735

[4] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2021. Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable. *ACM Trans. Parallel Comput.* 8, 1, Article 4 (April 2021), 70 pages. https://doi.org/10.1145/3434393

[5] Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.* 34, 3, Article 12 (May 2008), 25 pages. https://doi.org/10.1145/1356052.1356053

[6] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence. arXiv:2401.14196 [cs.SE] https://arxiv.org/abs/2401.14196

[7] Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. arXiv:2308.00352 [cs.AI] https://arxiv.org/abs/2308.00352

[8] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2021), 535–547. https://doi.org/10.1109/TBDATA.2019.2921572

[9] David Leopoldseder, Roland Schatz, Lukas Stadler, Manuel Rigger, Thomas Würthinger, and Hanspeter Mössenböck. 2018. Fast-path loop unrolling of non-counted loops to enable subsequent compiler optimizations. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes* (Linz, Austria) *(ManLang '18)*. Association for Computing Machinery, New York, NY, USA, Article 2, 13 pages. https://doi.org/10.1145/3237009.3237013

[10] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[11] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! arXiv:2305.06161 [cs.CL] https://arxiv.org/abs/2305.06161

[12] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (Dec. 2022), 1092–1097. https://doi.org/10.1126/science.abq1158

[13] Zhihao Li, Haipeng Jia, Yunquan Zhang, Tun Chen, Liang Yuan, Luning Cao, and Xiao Wang. 2019. AutoFFT: a template-based FFT codes auto-generation framework for ARM and X86 CPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 25, 15 pages. https://doi.org/10.1145/3295500.3356138

[14] Xiaoyan Liu, Xinyu Yang, Kejie Ma, Shanghao Liu, Kaige Zhang, Hailong Yang, Yi Liu, Zhongzhi Luan, and Depei Qian. 2024. Moirae: Generating High-Performance Composite Stencil Programs with Global Optimizations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Atlanta, GA, USA) *(SC '24)*. IEEE Press, Article 20, 15 pages. https://doi.org/10.1109/SC41406.2024.00026

[15] Jack L. Lo and Susan J. Eggers. 1995. Improving balanced scheduling with compiler optimizations that increase instruction-level parallelism. *SIGPLAN Not.* 30, 6 (June 1995), 151–162. https://doi.org/10.1145/223428.207132

[16] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. arXiv:2306.08568 [cs.CL] https://arxiv.org/abs/2306.08568

[17] Daniel Nichols, Joshua H Davis, Zhaojun Xie, Arjun Rajaram, and Abhinav Bhatele. 2024. Can large language models write parallel code?. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*. 281–294.

[18] Daniel Nichols, Aniruddha Marathe, Harshitha Menon, Todd Gamblin, and Abhinav Bhatele. 2024. HPC-Coder: Modeling Parallel Programs using Large Language Models. In *ISC High Performance 2024 Research Paper Proceedings (39th International Conference)*. IEEE, 1–12. https://doi.org/10.23919/isc.2024.10528929

[19] Daniel Nichols, Aniruddha Marathe, Harshitha Menon, Todd Gamblin, and Abhinav Bhatele. 2024. HPC-Coder: Modeling Parallel Programs using Large Language Models. In *ISC High Performance 2024 Research Paper Proceedings (39th International Conference)*. Prometeus GmbH, 1–12.

[20] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] https://arxiv.org/abs/2303.08774

[21] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir 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. arXiv:2105.12655 [cs.SE] https://arxiv.org/abs/2105.12655

[22] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL] https://arxiv.org/abs/2308.12950

[23] Ryuichi Sai, John Mellor-Crummey, Jinfan Xu, and Mauricio Araya-Polo. 2024. Automated Code Generation of High-Order Stencils for a Dataflow Architecture. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Atlanta, GA, USA) *(SC '24)*. IEEE Press, Article 19, 13 pages. https://doi.org/10.1109/SC41406.2024.00025

[24] Adrian Schmitz, Semih Burak, Julian Miller, and Matthias S. Müller. 2024. Parallel Pattern Compiler for Automatic Global Optimizations. *Parallel Comput.* 122 (2024), 103112. https://doi.org/10.1016/j.parco.2024.103112

[25] Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, 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)*.

[26] Tyler M. Smith, Robert van de Geijn, Mikhail Smelyanskiy, Jeff R. Hammond, and Field G. Van Zee. 2014. Anatomy of High-Performance Many-Threaded Matrix Multiplication. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 1049–1059. https://doi.org/10.1109/IPDPS.2014.110

[27] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. 2024. A survey on large language model based autonomous agents. *Frontiers of Computer Science* 18, 6 (2024), 186345.

[28] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024. Magicoder: Empowering Code Generation with OSS-Instruct. arXiv:2312.02120 [cs.CL] https://arxiv.org/abs/2312.02120

[29] Du Wu, Jintao Meng, Wenxi Zhu, Minwen Deng, Xiao Wang, Tao Luo, Mohamed Wahib, and Yanjie Wei. 2024. autoGEMM: Pushing the Limits of Irregular Matrix Multiplication on Arm Architectures. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15. https://doi.org/10.1109/SC41406.2024.00027

[30] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent J. Hellendoorn. 2022. A Systematic Evaluation of Large Language Models of Code. arXiv:2202.13169 [cs.PL] https://arxiv.org/abs/2202.13169

A Appendix

Listing 1. Prompt for RAG

```
You are a coding expert who writes very fast and highly optimized C/C++ code, specialized
in tools such as **OpenMP** for parallelization. Your goal is to rewrite serial code to make
it fast through parallel processing. The user will provide you with an original code snippet.

## Directions
Please adhere to the following instructions for your response:
1. Understand the Purpose
2. Analyze for Optimization
3. Provide Optimized Code

## Reference Examples:
Slow code 1: {slow_code1}
Fast code 1: {fast_code1}

Slow code 2: {slow_code2}
Fast code 2: {fast_code2}

##Source Code:
{source_code}

##Optimized Code:
{optimized_code}
```