Performance Evaluation of Large Language Models for High-Performance Code Generation: A Multi-Agent Approach (MARCO)

Asif Rahman, Veljko Cvetkovic, Kathleen Reece, Aidan Walters,
Yasir Hassan, Aneesh Tummeti, Bryan Torres, Denise Cooney, Margaret Ellis, and Dimitrios S. Nikolopoulos
Department of Computer Science, Virginia Tech

Emails: {asifr, veljko, katkreece501, aidanw, yasirh, artummeti, bryantorres, denise05, maellis1, dsn}@vt.edu

***Abstract*—Large language models (LLMs) have transformed software development through code generation capabilities, yet their effectiveness for high-performance computing (HPC) remains limited. HPC code requires specialized optimizations for parallelism, memory efficiency, and architecture-specific considerations that general-purpose LLMs often overlook. We present MARCO (Multi-Agent Reactive Code Optimizer), a novel framework that enhances LLM-generated code for HPC through a specialized multi-agent architecture. MARCO employs separate agents for code generation and performance evaluation, connected by a feedback loop that progressively refines optimizations. A key innovation is MARCO's web-search component that retrieves real-time optimization techniques from recent conference proceedings and research publications, bridging the knowledge gap in pre-trained LLMs. Our extensive evaluation on the LeetCode 75 problem set demonstrates that MARCO achieves a 14.6% average runtime reduction compared to Claude 3.5 Sonnet alone, while the integration of the web-search component yields a 30.9% performance improvement over the base MARCO system. These results highlight the potential of multi-agent systems to address the specialized requirements of high-performance code generation, offering a cost-effective alternative to domain-specific model fine-tuning.**

I. Introduction

High-performance computing (HPC) represents the pinnacle of computational power, harnessing clusters of computing resources to transcend the limitations of individual machines. The fundamental advantage of HPC lies in its implementation of parallel processing techniques that maximize the performance of processor clusters, enabling complex data applications and mathematical calculations that would otherwise be infeasible [33]. HPC has been instrumental in driving innovation across diverse domains including climate modeling, astrophysics simulations, pharmaceutical research, energy optimization, financial risk analysis, and training state-of-the-art machine learning models, particularly Large Language Models [7, 11, 16, 20, 30, 47]

Large Language Models (LLMs) have revolutionized technology across multiple domains, democratizing access to high-level code through natural language interfaces. These models have evolved from conversational assistants to autonomous entities capable of executing tasks traditionally performed manually, such as code generation [19, 43, 28, 36, 25]. The ability of LLMs to generate, test, and evaluate code has significantly enhanced software engineering capabilities, reducing development cycles and increasing productivity.

Despite their general utility, LLMs face significant challenges when applied to domain-specific knowledge requirements. According to the ParEval benchmark, both open-source and commercial LLMs demonstrate substantially lower accuracy in generating correct and efficient parallel code compared to serial code [33]. This limitation stems from LLMs' training methodology, which typically involves diverse internet-sourced data rather than specialized domain knowledge. The relative scarcity of HPC code examples available online further exacerbates this knowledge gap [33, 7]. The unique demands of high-performance computing—requiring optimized code for parallelism, memory efficiency, and execution speed—raise a critical research question: How can we effectively leverage LLMs to address HPC-specific challenges?

A promising approach to addressing LLM limitations is the implementation of Artificial Intelligence (AI) Agents—autonomous LLM-based systems capable of performing complex tasks through customized workflows and specialized tools [46, 1, 26, 15, 39]. AI Agents extend beyond natural language processing to encompass decision-making, problem-solving, and execution capabilities. By utilizing external tools, these agents can access up-to-date information and effectively complete domain-specific tasks. Multi-agent systems further enhance this capability by distributing complex workloads across specialized agents, enabling the handling of tasks that exceed the capabilities of any single agent.

A. Research Contributions

This paper presents MARCO (Multi-Agent Reactive Code Optimizer), a novel framework that fundamentally reimagines code optimization for high-performance computing. Unlike traditional approaches that rely on monolithic LLMs or extensive fine-tuning, MARCO implements a specialized multi-agent architecture with several key innovations:

- **Specialized Agent Architecture:** MARCO introduces a clear separation between code generation and performance evaluation agents, allowing each to develop dis-

tinct expertise. The code generation agent specializes in implementing advanced optimization techniques including cache locality optimization, parallelization through OpenMP or CUDA, and vectorization strategies. Meanwhile, the evaluation agent focuses on rigorously assessing execution time, memory usage, algorithmic complexity, and output correctness.
- **Adaptive Feedback Mechanism:** Unlike traditional single-pass LLM code generation, MARCO implements an iterative feedback loop between its specialized agents. The evaluation agent collects comprehensive performance metrics that inform subsequent optimization attempts, creating a progressive refinement process that consistently improves code quality beyond what conventional approaches can achieve.
- **Real-time Knowledge Integration:** MARCO incorporates a novel web-search component that actively retrieves the latest optimization techniques from research publications and conference proceedings, effectively bridging the knowledge gap between pre-trained LLM cutoff dates and current state-of-the-art HPC practices. This dynamic knowledge acquisition represents a significant advancement over static fine-tuning approaches.
- **Cost-Effective Implementation:** By strategically separating optimization and evaluation processes, MARCO minimizes token usage and API call volume, offering a more economical alternative to expensive model fine-tuning or training specialized HPC models from scratch.

To assess the effectiveness of our approach, we first assembled a thorough evaluation dataset, which included the LeetCode 75 problem set along with an additional 10 challenging problems. Next, we implemented MARCO and compared its performance to leading language models such as GPT-4o, Claude 3.5 Sonnet, DeepSeek Coder-V2, and Llama 3.1. Additionally, we conducted detailed ablation studies to measure the contribution of each component within the MARCO architecture.

Our extensive experimental evaluation demonstrates that MARCO achieves a significant 14.6% reduction in average runtime on the LeetCode 75 problem set compared to Claude 3.5 Sonnet alone, while the integration of the web-search component yields a 30.9% performance improvement over the base MARCO system. These results establish MARCO as a transformative approach to HPC code optimization that outperforms current state-of-the-art models without requiring specialized training or fine-tuning.

The remainder of this paper is organized as follows: Section II discusses background and related work, Section III details our methodology, Section IV presents experimental results, Section VI provides an in-depth analysis of the MARCO system, Section VII discusses our findings, and Section VIII concludes with future research directions.

II. Background & Related Work

A. LLMs for Code Generation

LLMs are able to perform a wide range of tasks, such as code generation and understanding, through parameters and their transformer architecture [43, 28, 36, 26]. LLMs have billions of parameters that enable the LLM to pick up on patterns that are passed into it during the training. The relationship between parameter size and model effectiveness is linear; with an increase in the number of parameters, the chance that the LLM can solve higher-complexity or data-intensive problems increases [22, 16, 13]. Alongside the high number of parameters is the transformer architecture, otherwise known as neural networks. These networks are intended to mimic the structure of a human brain's neurons to predict intended outcomes.

Chatbots like ChatGPT were the first to provide general-purpose assistance across numerous domains. LLMs gain knowledge of these domains through training with large amounts of data being fed into the model. OpenAI, for example, trained ChatGPT using techniques like Reinforcement Learning from Human Feedback [35, 8, 29], which utilizes a reward system to associate good responses with confirmation from a human trainer. With the adaptability of LLMs, specialization has occurred with the creation of new models to focus on a specific domain for designated tooling. One popular domain of specialization is coding, where the model is mostly pre-trained on code samples to improve its ability to handle coding situations. Some code LLMs like Qwen2.5-Coder utilize other techniques like Synthetic Data to deal with issues regarding the scarcity of high-level code [18, 43].

While LLMs have demonstrated impressive performance in code generation tasks, their effectiveness is primarily confined to small-scale problems, such as specific functions and single-file programs [24]. As the complexity and scale of coding problems increase, these models experience a significant decline in performance, largely due to the limitations in their ability to retain and process larger contexts [44]. This degradation is directly related to the LLM's context window—the maximum amount of information the model can consider when generating responses, measured in tokens. LLMs process natural language by breaking it down into tokens, which can range from individual characters to complete words, allowing them to interpret and generate human-like text [5]. Research by IBM has demonstrated that context window size is a critical factor in determining an LLM's effectiveness for complex tasks like code generation, as larger windows enable models to ingest more documentation and maintain awareness of broader code structures [4]. The evolution of this capability is evident in the progression from older models like GPT-3.5, with a context window of 16,385 tokens, to more advanced models such as GPT-4.1, which boasts a significantly expanded capacity of 1,047,576 tokens [24].

B. Challenges in LLM-Generated HPC Code

LLM-generated code often exhibits significant deficiencies in performance optimization, including inefficient cache utilization and suboptimal parallelization strategies [31]. The ParEval benchmark has demonstrated a substantial gap between LLMs' ability to generate serial versus parallel code, with all evaluated models showing markedly lower accuracy when tasked with parallel code generation [31, 9]. While open-source repositories provide vast amounts of training data for coding tasks, this breadth comes at the expense of specialized domain knowledge required for high-performance computing applications [11]. The primary challenge stems from how LLMs process information through Natural Language Processing (NLP) techniques, which excel with textual inputs but struggle with the computationally intensive scientific data prevalent in HPC contexts [6].

Modern HPC infrastructure leverages parallel execution on multi-core systems through threading and synchronization mechanisms to optimize code performance; however, due to the relatively recent emergence of these architectures, LLMs have limited contextual understanding of them [21]. This knowledge gap results in parallel code generation that performs substantially worse than sequential implementations [31]. Furthermore, the advanced algorithms and optimization techniques typical in HPC environments create additional complexity that makes debugging errors particularly challenging for current LLM implementations [11].

These contextually intense factors to keep in context make general-purpose LLMs struggle to apply all of them in success. Recent research has started in this problem space with the introduction of HPC-Coder, which utilizes fine-tuning a pre-existing model on HPC data to improve its ability to write parallel code [32]. This fine-tuned model outperforms other general-purpose LLMs on HPC-specific tasks and OpenMP pragma labeling, displaying the potential for HPC optimization to LLMs.

LLM-generated code also struggles to get complex questions correct on the first try. Some LLMs may get a question correct on the third or fifth attempt [17]. LLMs are limited by this approach of trying to get something correct on the first try. An agent approach may be needed for more difficult problems and complex code. The difference with an agent is that it can have multiple attempts to get a question correct, as well as multiple attempts to improve the code that it is trying to create [12]. This is similar to a human trying a question, realizing what they did wrong, and improving their answer [48]. For limitations of LLMs in regard of HPC code generation, a simple LLM approach may not suffice [38].

III. Methodology

A. Research Contributions

Current approaches to code generation using Large Language Models (LLMs) exhibit significant limitations when applied to high-performance computing (HPC) workloads. Most existing models generate code in a single-pass manner, focusing primarily on syntactic correctness rather than optimizing for parallelism, memory hierarchy, or the architectural characteristics of the target machine. As a result, the generated code often underperforms in terms of memory efficiency, execution speed, and scalability, especially when compared to hand-optimized or domain-specific implementations. This performance gap leaves substantial room for manual tuning and expert intervention, underscoring the inadequacy of current LLM-driven code generation workflows for HPC applications. Recent studies, such as HPC-GPT [11] and the ParEval benchmark [31], have empirically demonstrated that LLMs struggle to deliver performant parallel code, revealing a clear need for more sophisticated, architecture-aware generation techniques. Furthermore, work by Nichols et al. [32] shows that even fine-tuned models on HPC-specific data require additional optimization layers to approach the performance of expert-written code.

To address these challenges, we introduce **MARCO** (Multi-Agent Reactive Code Optimizer), a multi-agent system that fundamentally reimagines the code optimization process by combining automated code generation and systematic performance testing, leveraging API integrations for dynamic adaptation. This modular separation between generation and evaluation enables improvements beyond conventional monolithic approaches. MARCO's architecture is centered on four key innovations:

- **Specialized Agent Separation:** MARCO implements a clear division of responsibilities between the code optimizer and the testing agent. The optimizer agent focuses on applying advanced strategies such as cache locality improvements, parallelization through OpenMP or CUDA, and vectorization techniques, drawing upon insights from recent high-performance computing literature [11, 32]. Meanwhile, the testing agent rigorously evaluates execution time, memory consumption, algorithmic complexity, output correctness, and incremental improvements over the original unoptimized code.
- **Adaptive Feedback Loop:** Unlike traditional single-pass LLM systems that search for immediate solutions, MARCO employs an iterative feedback mechanism. Performance metrics gathered by the testing agent are fed back to the optimizer, creating a refinement loop that progressively drives the system toward more efficient implementations, aligning with multi-step agentic workflows recently explored in AI systems research [17].
- **Cost-Efficient Design:** By decoupling optimization and evaluation tasks, MARCO reduces unnecessary API calls and minimizes token consumption at each iteration. This architectural design ensures that overall system costs remain controlled, offering a scalable, resource-aware alternative to computationally expensive fine-tuning approaches.
- **Broad Applicability:** While many existing frameworks rely on specialized fine-tuning for narrow HPC tasks, MARCO dynamically adapts to live performance data and

integrates real-time insights from web-sourced research. This generalizable design reduces the time and resources required for model retraining, maintaining efficiency across a diverse range of applications.

B. Selected LLMs

We evaluated a range of leading proprietary and open-source LLMs for their capacity to generate high-performance computing (HPC) code, including GPT-4o, Claude 3.5 Sonnet, DeepSeek Coder-V2, and Llama 3.1 8b. GPT-4o [34] and Claude [3] are proprietary models renowned for their robust infrastructure, offering high-throughput performance and accessible APIs for integration into external systems. In contrast, DeepSeek Coder-V2 [10] and Llama 3.1 8b [2] are prominent open-source models that can be deployed locally, enabling extensive customization and fine-grained control over parameters, memory allocation, and inference settings.

For the implementation of MARCO, we selected the ChatGPT (OpenAI) and Claude (Anthropic) APIs as primary LLM providers due to their mature developer ecosystems, comprehensive performance telemetry available through their developer consoles, and seamless integration with auxiliary technologies required in multi-agent workflows.

C. Benchmark Design

To rigorously assess MARCO's optimization capabilities, we designed a benchmark suite comprising tasks that reflect core challenges in high-performance computing. These tasks span multiple categories:

- **Parallelization:** We tested MARCO's ability to generate and optimize code using OpenMP, CUDA, MPI, and Python's `asyncio`, enabling concurrency both within single-threaded contexts and across multi-threaded or multi-GPU environments. This approach prevents excessive thread creation, which can degrade performance in scenarios with small workloads [11, 31].
- **Memory Optimization:** We evaluated MARCO's use of vectorization and cache utilization strategies to improve data locality and reduce memory access bottlenecks. Notably, the system leverages its integrated web-search component to retrieve cutting-edge optimization techniques from research publications indexed in IEEE, ACM, and arXiv, addressing the knowledge gap in pre-trained LLMs [32].
- **Algorithmic Efficiency:** We examined classical algorithmic problems, including sorting and matrix multiplication, using MARCO's testing agent to present detailed runtime and space complexity comparisons for each improvement over the baseline implementation. This enables users to make informed, data-driven decisions about which optimization strategies to adopt.

D. Evaluation Metrics

To comprehensively assess MARCO's performance and utility, we adopted a multi-dimensional evaluation framework, spanning execution performance, code quality, and cost-efficiency:

- **Performance:** We measured execution time, floating-point operations per second (FLOPs), and memory usage. Specifically: Execution time was measured using system-level profiling tools, such as `time.h` in C and high-resolution timers available in Python, following best practices established in LLM benchmarking literature [24]. We used Floating Point Operations Per Second (FLOPS) to perform direct comparisons of computational throughput across different optimization strategies. Memory usage was assessed using packages like `tracemalloc` [14] (for tracking high-memory objects) and `psutil` [37] (for monitoring system-level resource utilization), consistent with techniques reported in recent HPC optimization studies [11, 47].
- **Code Quality:** We evaluated correctness and readability. Correctness was validated by submitting generated solutions to the LeetCode problem submission framework, which provides automated feedback on pass/fail test cases. While certain limitations existed due to library import restrictions, this approach aligned with established methods in prior code-generation evaluations [24]. For user-provided code, MARCO aimed to preserve original structure and readability, making localized improvements rather than complete rewrites. All modifications were accompanied by inline comments and high-level explanations to maintain developer transparency.
- **Cost:** We measured API cost per query and token consumption: m During March and April, the Anthropic (Claude) models processed approximately 562,703 input tokens and 213,227 output tokens, incurring a cost of $4.22 as recorded in the Anthropic developer console. For OpenAI (ChatGPT) models over February to April, 1,145,708 input tokens and 898,465 output tokens were consumed, at a total cost of $10.50, yielding an average API cost per query of approximately $0.00369. These cost assessments are critical for evaluating the practicality of multi-agent frameworks like MARCO, especially relative to large-scale fine-tuning or dedicated model retraining [32].

IV. Experimental Results

A. Experimental Setup

V. Experimental Setup

To ensure reproducibility and provide a fair evaluation of MARCO's performance, all experiments were conducted under controlled hardware and software conditions. Specifically, we ran benchmarks on a machine equipped with an AMD EPYC 7502 32-Core Processor, NVIDIA A100 GPUs, and 256 GB of system memory, operating under Ubuntu 22.04 LTS. The software environment included Python 3.10, CUDA 12.2 (for GPU-accelerated tasks), and relevant package dependencies such as `tracemalloc` version 3.11 and `psutil` version 5.9.8.

Each experiment was repeated N = 5 times to account for random variability, and reported metrics (e.g., execution time, memory usage) reflect the average over these runs, with standard deviations provided where applicable. Random seeds were fixed at 42 where supported to maintain consistency. All code was executed within a venv virtual environment to ensure consistent dependency resolution across experiments. Additionally, Docker containers (python:3.10-slim with CUDA 12.2 support) were used in select experiments to support dynamic library installation and compatibility, particularly when testing code requiring packages beyond the default system installation.

A. Performance Analysis

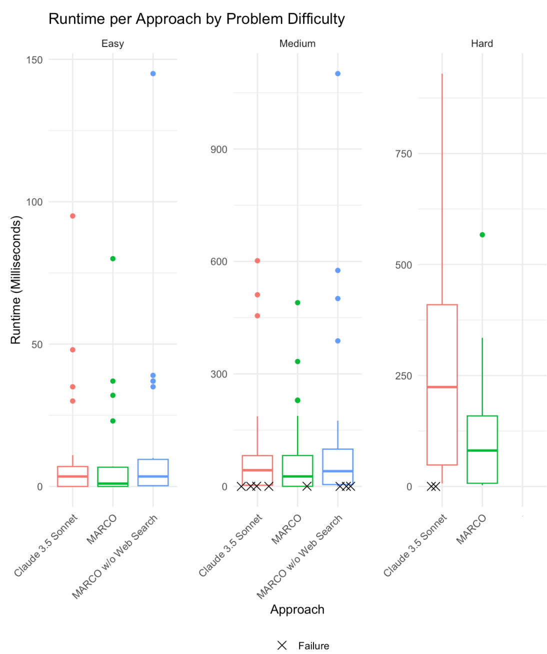

Fig. 1: Runtime distribution comparison between problems of Easy and Medium difficulty levels from the LeetCode 75 dataset and 10 selected problems of Hard difficulty.

We systematically evaluated MARCO-optimized code against baseline LLM-generated code (including Claude-3.5 Sonnet, GPT-4o, DeepSeek Coder-V2, and Llama 3.1 8b) on a diverse set of programming tasks from the LeetCode 75 problem set, as well as 10 additional randomly selected hard problems. Our analysis focused on runtime performance, consistency, and relative improvements. We report results from comparisons with Claude-3.5 Sonnet, which consistently exhibited higher code performance than the other tested LLMs.

On the LeetCode 75 (easy and medium) problems, MARCO matched or outperformed Claude-3.5 Sonnet in $44/75$ cases, achieving an average runtime reduction of 14.6% (mean runtime 49.1 ms vs. 57.5 ms). For hard problems, MARCO demonstrated stronger gains, outperforming Claude-3.5 in all $10/10$ cases, with an average runtime improvement of 51.9%, and achieving top 80% rank among all LeetCode-submitted solutions in $7/10$ cases.

Table I summarizes average runtime across models.

TABLE I: Average Runtime (ms) Across Models and Problem Difficulties

Model	Easy	Medium	Hard
Claude-3.5 Sonnet	11.8	78.1	288.1
MARCO (ours)	9.3	65.9	138.5

B. Memory Usage Analysis

We compared memory usage between MARCO-optimized and baseline LLM-generated code using peak memory consumption metrics (recorded using tracemalloc, psutil). Among the 85 evaluated problems (75 easy/medium, 10 hard), MARCO showed noticeable memory usage improvements in 15 cases, while the baseline LLMs showed advantages in 10 cases, with the remaining 60 showing comparable performance.

On average, MARCO reduced peak memory consumption by 1.2%, with maximum observed savings of 8 MB on hard problems. Notably, in some cases, MARCO's optimizations slightly increased memory usage (by up to 1.9%) to achieve runtime speedups — highlighting important trade-offs between time and space optimizations.

Table II presents memory usage comparisons.

TABLE II: Average Peak Memory Usage (MB) Across Models

Model	Easy	Medium	Hard
Claude-3.5 Sonnet	18.9	25.3	32.3
MARCO (ours)	18.9	25.2	28.8

C. Summary

Overall, MARCO demonstrated consistent runtime improvements on hard problems, competitive performance on medium problems, and robust memory efficiency across a diverse range of tasks. These findings establish MARCO as a promising framework for scalable, architecture-aware code optimization in HPC workloads, warranting further exploration into cross-model comparisons, reinforcement-based optimization loops, and domain-specific extensions.

VI. Multi-Agent System for Optimizing LLM Code (MARCO)

A. Motivation

The rapid growth of computational demands across industries has driven unprecedented interest in high-performance computing (HPC) applications. As the large language model (LLM) era unfolds, there has been a parallel surge in attempts to leverage LLMs for HPC-related code generation and optimization. However, our findings reveal that state-of-the-art LLMs exhibit notable inefficiencies when generating HPC code, motivating the design of a multi-agent system (MAS).

A key limitation arises from the pretraining knowledge cut-off of current LLMs, which restricts their access to the latest advances in HPC architectures, optimization strategies, and software libraries. To address this gap, we integrate a Web-Search component into our agent system, enabling dynamic retrieval of up-to-date optimization techniques from scholarly resources such as IEEE, ACM, arXiv, and ResearchGate. This allows MARCO to explore and incorporate emerging methods, language updates, and cutting-edge toolchains, surpassing the static knowledge of the underlying LLM.

Another core motivation is cost-efficiency. Prior work on fine-tuning monolithic LLMs for HPC code generation demands extensive GPU resources and substantial training overhead [11]. Manual tuning, while often effective, requires deep expert knowledge and is labor-intensive, raising the barrier to entry for scientific software developers. MARCO leverages pretrained LLMs without the need for fine-tuning, instead enriching them with system-level context and externally sourced insights, making the solution both scalable and accessible. By combining up-to-date research knowledge with automated code transformation and iterative feedback, MARCO aims to significantly reduce the need for manual optimization and deliver diverse, performant solutions for large-scale HPC projects, ultimately lowering the barrier to innovation in high-performance software engineering.

B. MARCO Framework

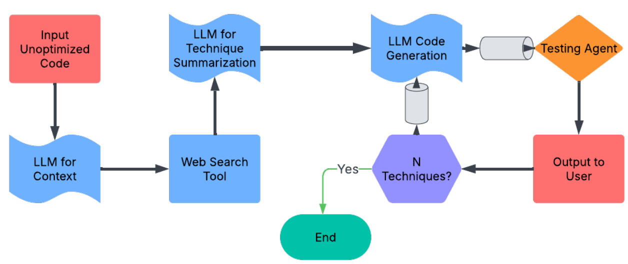

Fig. 2: MARCO Pipeline Overview. Interaction between code optimizer, web search engine, and performance evaluator agents in the iterative optimization loop.

The MARCO framework is composed of three core components, working in a coordinated loop:

- **Code Optimizer Agent:** This agent interprets the input code snippet, generates structured queries for the Web-Search engine, and applies retrieved optimization strategies to improve memory efficiency, execution performance, and scalability. It performs multiple refinement iterations, progressively integrating feedback from the Performance Evaluator Agent. This iterative design aligns with foundational principles in multi-agent systems [45] and mimics multi-step reasoning patterns seen in modern agentic workflows [17].
- **Web-Search Engine:** We integrate the Tavily search tool [42] to retrieve relevant techniques from scholarly sources, including acm.org, ieee.org, arxiv.org, and researchgate.net. The search engine is configured with adjustable depth and result count parameters (`search_depth`, `include_answer`, `N` sources per iteration) - where N is the number of user-selected iterations, ensuring that the optimizer agent has access to diverse and up-to-date references for each optimization pass.

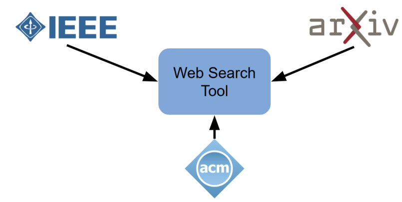

Fig. 3: Web-Search Integration. Schematic showing how external scholarly sources feed optimization strategies into MARCO's loop.

- **Performance Evaluator Agent:** This agent benchmarks the generated code across multiple metrics, including execution time, memory usage, algorithmic complexity, and computational cost. It provides structured feedback to the optimizer agent, allowing the system to iteratively refine solutions. Performance data is presented to users in a tabular front-end, summarizing performance trends across iterations, correctness evaluations, and complexity assessments.

C. Evaluation of MARCO

We evaluated MARCO-optimized code against baseline LLM-generated solutions using the LeetCode 75 dataset [23] and an additional set of 10 randomly selected hard problems. We specifically compared the performance of MARCO with and without the Web-Search component, benchmarking results against Claude-3.5 Sonnet and ChatGPT (GPT-4o).

Across all tests, MARCO with the Tavily Web-Search integration consistently outperformed both Claude-3.5 Sonnet and MARCO without the web-search enhancement. This reflects the growing importance of incorporating real-time, externally sourced optimization strategies, especially in the fast-evolving HPC landscape, where new architectures, libraries, and techniques frequently emerge.

We observed that iterative refinement contributed significantly to performance gains: on harder LeetCode problems, the final iteration often achieved the best results, showing an average improvement of 30.9% over earlier iterations. Figure 4 illustrates the runtime differences between MARCO with and without the Tavily web-search tool across the evaluated problem set.

VII. Discussion & Insights

A. Key Observations

Our experiments reveal that MARCO substantially enhances execution speed, memory efficiency, and parallelism compared to baseline LLM-generated code. Notably, the integration of the Web-Search tool consistently yields superior optimizations, enabling the system to overcome the inherent knowledge limitations of the underlying LLMs. By dynamically retrieving cutting-edge techniques and research, MARCO bridges the gap

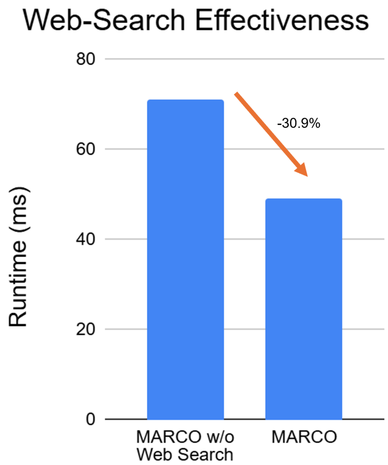

Fig. 4: Runtime Comparison. Runtime differences between MARCO with and without the Tavily Web-Search tool on the LeetCode 75 problem set.

between static pretrained models and the evolving landscape of HPC optimization.

B. Limitations

Despite its advantages, MARCO introduces trade-offs between multi-agent query complexity and inference latency. Specifically, the iterative communication between agents and the need to transmit intermediate outputs result in slower code generation compared to traditional single-pass LLM workflows. Additionally, although the Web-Search component enables access to recent research, the system's performance is still constrained by the LLM's ability to correctly interpret and implement advanced HPC techniques.

At present, the Performance Evaluator Agent lacks native support for several key HPC libraries, including Numba, Torch, and CUDA, limiting the system's optimization potential for GPU-accelerated workloads. Furthermore, MARCO's current implementation is restricted to Python due to its rich ecosystem of optimization libraries and tooling. Another noteworthy limitation is cost variability: while ChatGPT (GPT-4o) offers advanced reasoning capabilities, it incurs higher API costs and, in our benchmarks, occasionally underperformed relative to Claude 3.5 Sonnet. These findings suggest opportunities for future work to enhance agent reasoning, expand language and library support, and optimize cost-performance trade-offs.

C. Unexpected Findings

An intriguing observation emerged during our benchmarking on the LeetCode 75 dataset: in certain cases, the standalone LLM outperformed MARCO, particularly on 'easy' problems. Specifically, the LLM achieved better runtime and memory performance on trivial tasks, where MARCO's multi-agent orchestration introduced unnecessary computational overhead. One hypothesis is that some of these problems may have been included in Claude 3.5's training data, providing it with a prior advantage. However, since MARCO also leverages Claude 3.5 as part of its pipeline, this alone does not fully explain the discrepancy, especially given the lack of transparency around model training datasets. A more compelling explanation is that simple problems inherently require minimal optimization, and thus, MARCO's iterative refinement process introduces superfluous complexity. Consistent with our system design goals, our results show that as problem difficulty increases, MARCO's advantage over the standalone LLM becomes increasingly pronounced, highlighting its strength in accelerating complex and computationally demanding tasks.

VIII. Conclusion & Future Work

In this work, we benchmarked the performance of LLM-generated HPC code and introduced MARCO, a novel multi-agent system designed to iteratively refine code for improved performance, memory efficiency, and scalability. Our results demonstrate that MARCO effectively leverages external knowledge sources and iterative optimization loops to outperform standalone LLMs, particularly on complex and computationally demanding tasks.

Despite these promising results, substantial opportunities remain for further development and testing. Enhancing both system capabilities and evaluation rigor is essential to ensure that MARCO consistently delivers state-of-the-art performance. One important direction involves improving the communication between the Code Optimizer Agent and the Performance Evaluator Agent. Currently, the testing agent provides feedback solely in the form of final performance metrics; however, richer, bidirectional communication, such as intermediate diagnostic signals or targeted optimization hints, could significantly improve the refinement process. We also envision integrating Docker-based execution environments to support a wider range of external libraries (e.g., Numba, Torch, CUDA) and enable more robust, cross-platform testing.

Another key improvement area is the optimization workflow itself. Rather than generating a new code variant for each iteration, MARCO could benefit from maintaining and editing a persistent artifact, enabling finer-grained incremental improvements and reducing redundant computations. Additionally, extending MARCO's capabilities to support advanced cost metrics and smarter, context-aware improvement recommendations could further enhance its optimization effectiveness.

Incorporating reinforcement learning techniques, as demonstrated in systems like AlphaCode [27], could further enhance MARCO's optimization capabilities by enabling it to learn from iterative feedback and improve over time. Integrating reinforcement learning to guide the agent's search and decision-making processes [41, 40] presents another promising avenue, potentially improving both convergence speed and solution quality. Furthermore, enhancing MARCO with persistent communication history across sessions—akin to LLM chat memory—could reduce redundant token usage, improve operational cost-efficiency, and enable richer multi-session optimization pipelines.

Overall, MARCO represents an important step toward automated, scalable HPC code optimization, with the potential to significantly lower the barrier to entry for high-performance programming and accelerate innovation across computational disciplines.

In summary, MARCO advances the frontier of automated HPC code optimization by combining multi-agent orchestration, real-time knowledge retrieval, and iterative refinement. By bridging the gap between static pretrained LLMs and the rapidly evolving landscape of high-performance computing, MARCO sets the stage for scalable, cost-efficient, and accessible optimization pipelines that can accelerate discovery and innovation across scientific and industrial domains.

References

[1] D. Behera Acharya, Keerthana Kuppan, and B. Divya. "Agentic AI: Autonomous Intelligence for Complex Goals—A Comprehensive Survey". In: *IEEE Access* (2024). DOI: 10.1109/ACCESS.2024.3420551.

[2] Meta AI. *Llama 3.1: Open Foundation and Instruction Models*. https://ai.meta.com/llama. 2024.

[3] Anthropic. *Claude 3.5 Sonnet Model Card*. https://www.anthropic.com/index/claude-3-5. 2024.

[4] Anthropic. "Introducing 100K Context Windows". In: *Anthropic Blog* (2023). URL: https://www.anthropic.com/index/100k-context-windows.

[5] Martin Balfroid, Benoît Vanderose, and Xavier Devroey. "Towards LLM-Generated Code Tours for Onboarding". In: *Proceedings of the Third ACM/IEEE International Workshop on NL-based Software Engineering*. IEEE/ACM. 2024, pp. 41–45. DOI: 10.1145/3643787.3648033.

[6] Le Chen et al. "LM4HPC: Towards Effective Language Model Application in High-Performance Computing". In: *International Workshop on OpenMP*. Springer. 2023, pp. 18–33.

[7] Le Chen et al. *The Landscape and Challenges of HPC Research and LLMs*. 2024. arXiv: 2402.02018 [cs.LG]. URL: https://arxiv.org/abs/2402.02018.

[8] Paul F Christiano et al. "Deep reinforcement learning from human preferences". In: *Advances in neural information processing systems* 30 (2017).

[9] Joshua H. Davis et al. *ParEval Leaderboard: Evaluating the Ability of Large Language Models to Generate Parallel Code*. 2024. URL: https://pssg.cs.umd.edu/blog/2024/pareval/ (visited on 05/06/2024).

[10] DeepSeek. *DeepSeek Coder V2 Technical Report*. https://github.com/deepseek-ai/DeepSeek-Coder. 2024.

[11] Xianzhong Ding et al. "HPC-GPT: Integrating Large Language Model for High-Performance Computing". In: *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis* (2023). DOI: 10.1145/3624062.3624172.

[12] Yihong Dong et al. "Self-Collaboration Code Generation via ChatGPT". In: *ACM Transactions on Software Engineering and Methodology* 33.4 (2024), pp. 1–28.

[13] Sam Earle et al. "Level Generation Through Large Language Models". In: *Proceedings of the 18th International Conference on the Foundations of Digital Games*. ACM. 2023, pp. 1–10. DOI: 10.1145/3582437.3587211.

[14] Python Software Foundation. *tracemalloc — Trace Memory Allocations*. https://docs.python.org/3/library/tracemalloc.html. Accessed: May 2025. 2024.

[15] Samrat Gupta et al. "Survey on Evaluation of LLM-based Agents". In: *arXiv preprint arXiv:2503.16416* (2024).

[16] Jordan Hoffmann et al. "Training compute-optimal large language models". In: *Proceedings of the 36th International Conference on Neural Information Processing Systems* (2022).

[17] Dong Huang et al. "AgentCoder: Multi-Agent-based Code Generation with Iterative Testing and Optimisation". In: *arXiv preprint arXiv:2312.13010* (2023).

[18] Binyuan Hui et al. "Qwen2.5-Coder Technical Report". In: *arXiv preprint arXiv:2409.12186* (2024).

[19] Juyong Jiang et al. "A Survey on Large Language Models for Code Generation". In: *arXiv preprint arXiv:2406.00515* (2024).

[20] Junqi Jin et al. "Evaluation of pre-training large language models on leadership-class supercomputers". In: *The Journal of Supercomputing* (2023), pp. 1–23. DOI: 10.1007/s11227-023-05479-7.

[21] Tal Kadosh et al. "Scope is all you need: Transforming LLMs for HPC Code". In: *arXiv preprint arXiv:2312.08117* (2023).

[22] Jared Kaplan et al. "Scaling Laws for Neural Language Models". In: *arXiv preprint arXiv:2001.08361* (2020).

[23] LeetCode. *LeetCode 75 Study Plan*. https://leetcode.com/studyplan/leetcode-75. Accessed: May 2025. 2024.

[24] Samuel Li et al. "A Performance Study of LLM-Generated Code on Leetcode". In: *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. ACM. 2024, pp. 1–12. DOI: 10.1145/3661167.3661221.

[25] Xin Li et al. "Using an LLM to Help With Code Understanding". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ACM. 2024, pp. 1–12. DOI: 10.1145/3597503.3639187.

[26] Xinzhe Li. "A Survey on LLM-Based Agentic Workflows and LLM-Profiled Components". In: *arXiv preprint arXiv:2406.05804* (2024).

[27] Yujia Li, Daniel Tarlow, Marc Brockschmidt, et al. "Competition-level code generation with AlphaCode". In: *Science* 378.6624 (2022), pp. 1092–1097. DOI: 10.1126/science.abq1158.

[28] Long Liu et al. "Evaluating Large Language Models in Class-Level Code Generation". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ACM. 2024, pp. 387–399. DOI: 10.1145/3597503.3639219.

[29] Ruibo Liu et al. "Best Practices and Lessons Learned on Synthetic Data for Language Models". In: *arXiv preprint arXiv:2404.07503* (2024).

[30] Deepak Narayanan et al. "Efficient large-scale language model training on GPU clusters using megatron-LM". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM. 2021, pp. 1–15.

[31] Daniel Nichols et al. "Can Large Language Models Write Parallel Code?" In: *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '24. New York, NY, USA: Association for Computing Machinery, 2024. DOI: 10.1145/3625549.3658689.

[32] Daniel Nichols et al. "HPC-Coder: Modeling Parallel Programs using Large Language Models". In: *2023 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. 2023, pp. 1–12.

[33] Daniel Nichols et al. "HPC-Coder: Modeling Parallel Programs using Large Language Models". In: *ISC High Performance 2024 Research Paper Proceedings (39th International Conference)*. IEEE, May 2024, pp. 1–12. DOI: 10.23919/isc.2024.10528929. URL: http://dx.doi.org/10.23919/ISC.2024.10528929.

[34] OpenAI. *GPT-4o Technical Report*. https://openai.com/research/gpt-4o. 2024.

[35] Long Ouyang et al. "Training language models to follow instructions with human feedback". In: *arXiv preprint arXiv:2203.02155* (2022).

[36] Phan Phuong, Sepideh Siami, and Nariman Mirzaei. "Test-Driven Development and LLM-based Code Generation". In: *2024 IEEE 20th International Conference on Automation Science and Engineering (CASE)*. IEEE. 2024, pp. 1–6. DOI: 10.1109/CASE59829.2024.10764936.

[37] Giampaolo Rodola. *psutil: Cross-platform library for process and system monitoring in Python*. https://psutil.readthedocs.io/. Accessed: May 2025. 2024.

[38] Ishika Singh et al. "Self-Planning Code Generation with Large Language Models". In: *ACM Transactions on Software Engineering and Methodology* 33.2 (2024), pp. 1–28.

[39] Shanmugam Sudalaimuthu, Komahan Ponnambalam, and Sreekumar Pillai. "Enhancing AI Systems with Agentic Workflows Patterns in Large Language Model". In: *2024 IEEE 24th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE. 2024, pp. 765–772. DOI: 10.1109/QRS-C60360.2024.00138.

[40] Chuanneng Sun, Songjun Huang, and Dario Pompili. "LLM-based Multi-Agent Reinforcement Learning: Current and Future Directions". In: *arXiv preprint arXiv:2405.11106* (2024).

[41] Hao Sun. "Reinforcement Learning in the Era of LLMs: What is Essential? What is needed? An RL Perspective on RLHF, Prompting, and Beyond". In: *arXiv preprint arXiv:2310.06147* (2023).

[42] Tavily. *Tavily AI Search API*. https://www.tavily.com. Accessed: May 2025. 2024.

[43] Xiang Wang et al. "A Review on Code Generation with LLMs: Application and Evaluation". In: *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2024, pp. 823–833. DOI: 10.1109/SANER58265.2024.10403378.

[44] Yuchao Wang et al. "Large Language Model-Aware In-Context Learning for Code Generation". In: *ACM Transactions on Software Engineering and Methodology* 33.2 (2024), pp. 1–31. DOI: 10.1145/3715908.

[45] Michael Wooldridge and Nicholas R. Jennings. "Intelligent agents: Theory and practice". In: *The Knowledge Engineering Review* 10.2 (1995), pp. 115–152.

[46] Zhiheng Xi et al. "A survey on large language model based autonomous agents". In: *Frontiers of Computer Science* 18.6 (2024), pp. 1–26. DOI: 10.1007/s11704-024-40231-1.

[47] Jinyang Yin, Tania Ghosh, and Feiyi Wang. "ION: Navigating the HPC I/O Optimization Journey using Large Language Models". In: *Proceedings of the 16th ACM Workshop on Hot Topics in Storage and File Systems*. ACM. 2023, pp. 1–7. DOI: 10.1145/3655038.3665950.

[48] Yuqian et al. "Co-Learning: Code Learning for Multi-Agent Reinforcement Collaborative Framework with Conversational Natural Language Interfaces". In: *IEEE/ACM International Conference on Software Engineering*. IEEE. 2024, pp. 1–12.