Agentic AI vs ML-based Autotuning: A Comparative Study for Loop Reordering Optimization

Miguel Romero Rosas
University of Delaware
Newark, USA
miguelro@udel.edu

Rudolf Eigenmann
University of Delaware
Newark, USA
eigenman@udel.edu

Khaled Ibrahim
Lawrence Berkeley National Laboratory (LBNL)
Berkeley, USA
kzibrahim@lbl.gov

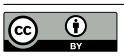
Abstract

High Performance Computing (HPC) applications rely heavily on code optimizations to achieve good performance on modern CPU and GPU architectures. Traditional Machine Learning autotuning approaches have demonstrated success in exploring high-dimensional spaces, but they often require expensive compile-run evaluations and lack adaptability for large HPC applications. The recent advances in Large Language Models (LLMs) and Agentic AI systems raise intriguing questions about the potential of these approaches to address specific optimization methodologies. This work aims to answer an essential question for the HPC community: "How Agentic AI Systems Compare to Traditional ML Autotuning Techniques?" To address this question, we present a comparative analysis between a traditional ML-based optimization approach and an Agentic AI system, evaluating their respective capabilities and limitations for loop-level optimization. In addition, we introduced a new Agentic AI system named LoopGen-AI using three different Large Language Models: GPT-4.1, Claude 4.0, and Gemini 2.5.

A key finding is that **LoopGen-AI achieves competitive performance with only a few program runs**, the reasoning logs from the agents revealed that their decisions rely heavily on the combination of semantic understanding of the target kernel with dynamic feedback from the environment, highlighting a promising new dimension in performance tuning. In contrast, ML-based autotuners focus on statistical exploration, and require orders of magnitude more runs to reach peak performance. Additionally, our analysis shows that prompt engineering, particularly using *Persona + Context Manager* patterns, significantly impacts the effectiveness of Agentic AI. Our results indicate that while Agentic AI systems are not yet a complete replacement for ML-based autotuners, it can effectively complement traditional methods.

CCS Concepts

• **Computing methodologies** → **Parallel programming languages**; **Classification and regression trees**; **Natural language generation**; **Feature selection**; • **Software and its engineering** → **Software development process management**.



SC Workshops '25, St Louis, MO, USA

ACM ISBN 979-8-4007-1871-7/25/11
https://doi.org/10.1145/3731599.3767401

Keywords

Agentic AI, Loop Reordering Optimization, Autotuning, ML-based optimization, Compilers and Large Language Models.

ACM Reference Format:
Miguel Romero Rosas, Rudolf Eigenmann, and Khaled Ibrahim. 2025. Agentic AI vs ML-based Autotuning: A Comparative Study for Loop Reordering Optimization. In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3731599.3767401

1 Introduction

High Performance Computing (HPC) applications are increasingly complex, and finding the correct set of optimizations that yields maximum performance is non-trivial. Achieving optimal performance often requires loop-level optimizations, such as loop interchange, unrolling, and tiling, which will directly impact cache locality, and parallel scalability [16]. Among these optimizations, loop reordering is critical because memory access patterns and cache reuse heavily influence execution time on modern architectures [5].

Traditional Machine Learning autotuning approaches have demonstrated success in exploring high-dimensional spaces. Tools such as GPTune [8], OpenTuner [4], HpBandSter [6] and ytopt [22] utilized supervised learning, Bayesian Optimization, or nobel strategies to find good parameter configurations through the use of surrogate models that relate tuning parameters to performance without the need to exhaustive exploration. However, these approaches can still require many runs of the application to achieve good performance.

Meanwhile, Large Language Models (LLMs) within an Agentic AI system offer a new horizon for code optimization. Agentic AI refers to an LLM-powered system capable of autonomous interaction with its environment, while iteratively proposing, testing, and refining new configurations [19]. Recent work illustrates their potential [7, 10–12, 14, 15, 18, 20, 23].

Despite the different advances in agentic AI, no prior work has compared an agentic AI system and ML-based autotuning approach for loop reordering optimization on high-performance scientific applications. ML-based autotuners rely on exploration and exploitation strategies focusing on statistical analysis to navigate unknown optimization spaces, but they cannot adapt to compilation errors, runtime anomalies, or unseen loop structures. In contrast, Agentic AI systems are equipped with semantic reasoning, environment feedback, and iterative correction, which may explore and optimize dimensional spaces without extensive offline learning.

In this paper, we make the following contributions:

- We conduct a comparison of a traditional ML-based autotuning vs an Agentic AI system on a real world case scenario.
- We introduce LoopGen-AI, an Agentic AI system that leverages GPT-4.1, Claude 4.0, and Gemini 2.5 to autonomously propose, refine, and validate loop reordering transformations.
- We provide insights for integrating LLM-driven agentic strategies into HPC performance engineering pipelines.

This paper is organized as follows: Section 2 describes both environment, ML-base autotuning and LoopGen-AI. Section 3 evaluates the capabilities of each environment and explores its limitations, followed by a discussion of related work in Section 4 and conclusions in Section 5.

2 Traditional ML-based Autotuning and Agentic AI System

This section presents the two optimization frameworks evaluated in this study: a traditional ML-based autotuning framework **GPTune**, and our proposed Agentic AI, **LoopGen-AI**. Section 2.1 explains the GPTune architecture and its strategy for exploring the loop transformation space, whereas Section 2.2 introduces LoopGen-AI, which leverages Large Language Models (LLMs) to iteratively generate, evaluate, and refine loop reordering strategies in an autonomous feedback driven system. Both systems **1. Navigate the search space**, then **2. Compile** and **3. Execute the code**. Their main difference lies in how they navigate the search space. By describing their respective workflows and architectures, we establish the basis for a fair comparative evaluation in the following sections.

2.1 GPTune: ML-based Autotuner

GPTune [8] is a traditional ML-based autotuning framework designed to optimize HPC application performance by exploring high-dimensional parameter spaces. Its workflow follows a typical exploration and exploitation strategy through statistical modeling and its decisions are driven primarily by numeric outcomes (runtime) rather than semantic meaning. GPTune samples a candidate parameter configuration, evaluates its performance on the target application, and uses a traditional machine learning model (Bayesian Optimization) to predict promising regions of the search space.

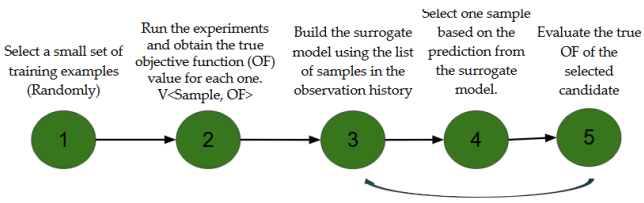


Figure 1: GPTune workflow

In the context of loop reordering optimization, we used GPTune to treat loop permutations as the tunable space. For each candidate configuration, GPTune generates the transformed kernel, compiles and executes it, and records its execution time, see Figure 1. The measured performance is then fed back into the ML model, enabling GPTune to guide the next subsequent searches toward different configurations expected to reduce runtime. Although this approach is effective, it requires a large number of runs, which makes it unsuitable for large HPC applications.

2.2 LoopGen-AI

LoopGen-AI is an agentic AI system for loop-reordering optimization. Unlike traditional autotuners, LoopGen-AI leverages LLMs such as GPT-4.1, Claude 4.0, and Gemini 2.5 to combine semantic understanding of the kernel with dynamic feedback from the environment. Its workflow consist of 4 components: **1. Preparation Phase**, **2. Action Phase**, **3. Compilation Phase** and **4. Execution Phase**, see Figure 2. These components will be explained in the following sections.

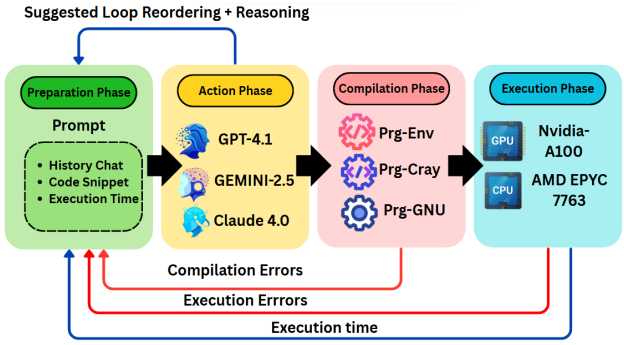


Figure 2: LoopGen-AI: Workflow.

2.2.1 ***Preparation Phase.*** This phase is responsible for constructing the prompt to be sent to each agent. Our prompt incorporates four state-of-the-art prompt patterns: ***1. Context Manager***, ***2. Persona***, ***3. Reflection***, *and* ***4. Template*** [21] [7] [11] [17].

These patterns enable each agent to: Focus on the relevant context of the application **(Context Manager)**. Assume a specific role, in this case the role of a compiler optimization expert **(Persona)**. Explain the reasoning behind its suggestions **(Reflection)**, and produce output in a precise JSON format for automated parsing **(Template)**. These patterns are applied in our prompt, see Figure 3.

The code snippet under evaluation consists of 6 levels of perfectly nested loops over **(is, ib, ms, mb, ns, nb)**, see Figure 4. Treating loop reordering as a permutation, the search space is 6! = 720 distinct loop orders.

Once the prompt has been constructed, we send it to three agents: GPT-4.1 [3], Gemini-2.5 [2], and Claude-4.0 [1]. Each agent is responsible for generating a new loop reordering and its respective reasoning process.

2.2.2 ***Action Phase.*** In this phase, each agent (**GPT-4.1, Gemini 2.5 and Claude 4.0**) analyzes the provided kernel and its context to propose a new loop reordering strategy. Each agent operates **independently**, using the contextual information provided in the prompt, which includes: **1. The loop structure** of the kernel under evaluation, **2. The History chat** including the reasoning of the respective agent in previous iterations. **3. The execution time** of the previous iteration, and any **4. Feedback or compilation errors** from prior attempts. Each agent produces an output in a

```
1. [Context Manager]: Start Over
2. [Persona]: Act as a compiler optimization expert specializing in loop
       transformations.
3. [Context Manager]: We are going to have an interactive conversation focused
       on optimizing loop ordering. I will provide you with a baseline code
       snippet and its execution time. Your task is to suggest a new loop
       ordering that reduces the execution time. Once you provide your
       suggestion, I will run the updated code and share the new execution time
       with you and the code snippet. The code under discussion consists of a
       6-level perfectly nested loop with loop variables is, ib, ms, mb, ns,
       and nb, each iterating over small, fixed bounds (Ns=6, Nb=4). The
       innermost loop updates a scalar variable tot_I using an OpenMP reduction
       and includes conditional logic with complex arithmetic. The loop body
       accesses three large arrays: G2xxxx and G2xyxy (both 1D vectors of size
       Ns^2 x Ns^2 x Nb^2 x Nb^2 = 331,776 complex<double> elements) via idx2 =
       IDX_8D(is,js,ib,jb,ns,ms,nb,mb), and array W via idx3 =
       IDX_8D(ns,ms,nb,mb,is,ls,ib,lb), where js, jb, ls, lb are outer loop
       variables.

#define IDX_8D(x,y,z,t,m,n,q,w) ((x) + (y)*Ns + ((z) + (t)*Nb)*Ns2 +
       ((m)+(n)*Ns)*Ns2*Nb2 + ((q) + (w)*Nb)*Ns2*Ns2*Nb2)

where Ns2 = Ns*Ns and Nb2 = Nb*Nb. The only data dependency is the reduction on
       tot_I; all other iterations are independent. The primary optimization
       goal is to maximize cache spatial locality. The target hardware is (CPU
       or GPU hardware info.....) running under the PrgEnv-(CRAY or
       Nvidia....) environment on the Perlmutter system at NERSC

4. [Reflection]: After each suggestion, briefly explain the reasoning and
       assumptions behind your recommendation, how you came up with the answer.

5. [Template]: Please respond in strict JSON format with the following keys:
{
 'suggested_parallel_loop': 'Your C++ loop goes here',
 'reasoning': 'Explain why you chose this loop structure and the approach you
       used',
}

Only output valid JSON. Do not include any markdown formatting.
```

Figure 3: Prompt Strategy

```
1: #pragma experimental start
2: for(int is = 0; is < Ns; ++is){
3:  for(int ib = 0; ib < Nb; ++ib){
4:    for(int ms = 0; ms < Ns; ++ms){
5:      for(int mb = 0; mb < Nb; ++mb){
6:        for(int ns = 0; ns < Ns; ++ns){
7:          for(int nb = 0; nb < Nb; ++nb){
8:              int idx2 = IDX_8D(is,js,ib,jb,ns,ms,nb,mb);
9:              int idx3 = IDX_8D(ns,ms,nb,mb,is,ls,ib,lb);
10:              if(is==ls && is==ns && is==ms && ib==lb &&
                     && ib==nb && ib==mb){
11:                  tot_I += -im*G2xyxy[idx2]*W[idx3];
12:              else{
13:            tot_I += -im*(G2xxxx[idx2]+G2xyxy[idx2])*W[idx3];
14:              }
15:             }
16:           }
17:         }
18:       }
19:     }
20:   }
21:  }
22: #pragma experimental end
```

Figure 4: Kernel extracted from an HPC application that calculates the Real Time Dyson Expansion (RT-DE). Parallelization relies on OpenMP pragmas on CPU and target offload on GPU.

strict JSON format. The output will include the new loop ordering and the reasoning behind its strategy.

2.2.3 ***Compilation Phase.*** The compilation Phase is responsible for validating and compiling the loop transformation produced in the Action Phase. After each agent generates its corresponding loop reordering, the code snippet is injected into the application source code and compiled using one of the two Perlmutter Programming environments (PrgEnvs) [9].

In this work, we used the following two PrgEnvs: **1. PrgEnv-NVIDIA**, it provides access to the Nvidida HPC SDK compilers (`nvc`, `nvc++`, `nvfortran`) optimized for both CPU and GPU execution. The other programming environment is **2. PrgEnv-CRAY**, it Offers the Cray Compiling Environment (CCE), which supports both CPU and GPU execution with OpenMP offloading and OpenACC.

During the compilation phase, any error is automatically captured and fed back to the preparation phase, see Figure 2. The prompt is then augmented with a concise history of prior failures, forming a critical component of the agentic feedback loop that allows LoopGen-AI to self correct in subsequent iterations.

2.2.4 ***Execution Phase.*** Once the kernel has been successfully compiled, the Execution Phase is initiated. In this phase, the generated binary is executed either on a CPU node (AMD EPYC 7763) or in a GPU node (Nvidia A100). Additionally, during the execution phase the system performs the following tasks: **1. Error monitoring:** Any runtime errors or anomalies are automatically captured and sent back to the Preparation Phase to inform the next iteration and **2. Performance measurement:** If execution completes successfully, the kernel execution time is measured and sent back to the preparation phase.

3 Experimental Setup

To evaluate the performance of our proposed Agentic AI system (LoopGen-AI) against the traditional ML-based autotuner (GPTune), we conducted experiments on the Perlmutter Supercomputer at the National Energy Research Scientific Computing Center (NERSC) [9]. Our experiments target a single real-world scientific application that computes the Real Time Dyson Expansion (RT-DE) [13], whose kernel was previously discussed in Section 2.2.1.

The experiments were conducted on Perlmutter's heterogeneous nodes using both CPU and GPU configurations. For the CPU, we used AMD EPYC 7763 (Zen3) with 32 cores and 512 GB of DDR4 memory per node, and for the GPU architecture, we used NVIDIA A100 (40 GB HBM2e). The compiled binaries were executed under two different programming environments (PrgEnvs) discussed in Section 2.2.3.

We evaluated two optimization strategies. For the Agentic AI (LoopGen-AI), we limited the runs to 10 iterations because further iterations showed only slight improvements. The best configuration is usually discovered within 3-7 iterations and the average execution time over those ten runs was reported. In contrast, the ML-based autotuning (GPTune) explored 360 candidate configurations, $\frac{1}{2}$ of the full $6! = 720$ search space because further configurations yielded <1% improvement. The best configuration was typically discovered within 150-250 iterations.

3.1 Impact of Prompt Pattern

As discussed in Section 2.2.1, the LoopGen-AI Preparation Phase utilizes prompt patterns to guide the reasoning of LLM agents. In our experiments, we evaluated the impact of the following three prompt

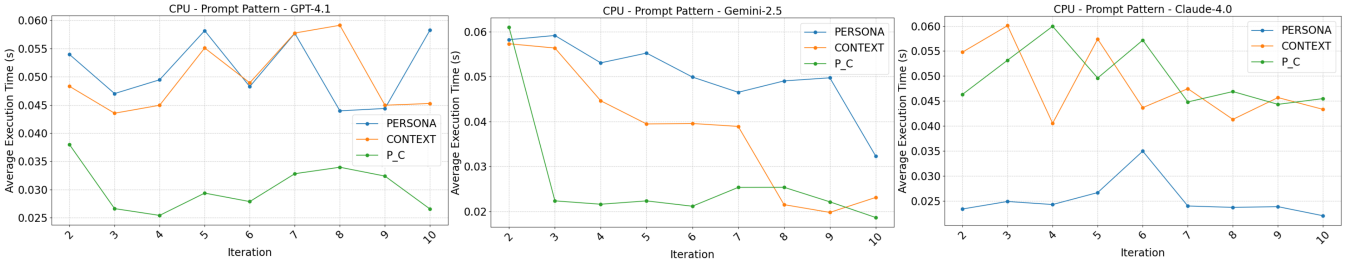


Figure 5: Prompt Pattern Performance on GPT-4.1, Gemini-2.5, and Claude-4.0.

strategies: **1. Persona**, **2. Context Manager**, and **3. Persona + Context Manager (P_C)**.

To assess the effectiveness of each prompt pattern, we measured the average kernel execution time across the two programming environments discussed in Section 2.2.2. The CPU results on the AMD EPYC 7763 are shown in Figure 5.

From the figure, we observe that the **Persona + Context Manager (P_C)** pattern often achieves the lowest execution time, indicating that combining expert role guidance with contextual information about the kernel and the type of architecture leads to more effective loop reordering optimizations.

The Persona pattern alone often underperforms, as the lack of explicit context can lead to suboptimal loop reorderings. The Context Manager pattern performs better than Persona in isolation, but benefits significantly from the combination with Persona.

3.2 Speedup evaluation for the ML based autotuning and Agentic AI system

To evaluate the performance improvement of the two optimization approaches, we computed the speedup of the optimized kernel relative to the baseline manually optimized on both CPU and GPU. The baseline corresponds to the manually optimized of the Real Time Dyson Expansion (RT-DE) kernel.

The speedup S is defined as:

$$S = \frac{T_{\text{baseline\_manually\_optimized}}}{T_{\text{automatically\_optimized}}}$$

where $T_{\text{baseline\_manually\_optimized}}$ represents the execution time of the manually optimized original kernel and $T_{\text{automatically\_optimized}}$ represents the average execution time achieved by either **Agentic AI (LoopGen-AI)** or **ML-based autotuning (GPTune)**.

Figure 6 shows the **GPTune** speedup distribution on CPU and GPU under the two programming environments discussed in Section 2.2.3. Figure 7–9 show the CPU speedup distribution for **LoopGen-AI** using three prompt patterns discussed in Section 2.2.1 with the respective LLMs (**GPT-4.1, Gemini 2.5 and Claude 4.0**). The corresponding GPU results are shown in Figure 10–12.

3.2.1 ***CPU Speedup Evaluation.*** Figure 6 shows that GPTune's effectiveness is environment-dependent. On **CRAY**, the distribution is centered well above baseline (median ~1.3–1.5×). On **NVIDIA (CPU mode)**, gains are smaller and more variable (median ~1.05–1.15×). GPTune delivers stable CPU gains on CRAY and smaller gains on NVIDIA-CPU.

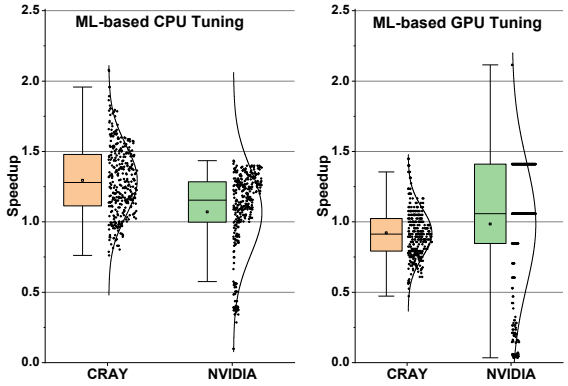


Figure 6: ML-Based autotuning for CPU and GPU: for two programming environments: CRAY, and NVIDIA

Figure 7 highlights a dependence on both the programming environment and the prompt pattern on GPT-4.1. **On CRAY (left)**, all three patterns deliver substantial gains (medians ~2.6–3.1×), with *Context* and *Context+Persona* outperforming *Persona*. **On NVIDIA (CPU mode, right)** *Context* and *Persona* alone are typically below baseline (< 1×), while *Context+Persona* is the only variant that consistently reaches around ~1.2–1.3×, but with higher variance.

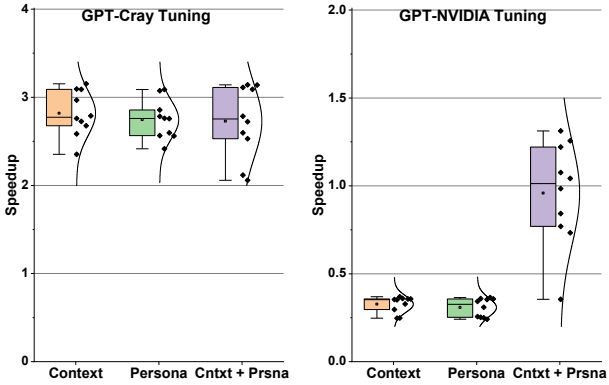


Figure 7: GPT-4.1-based Tuning: CPU Speedup for the two programming environments: CRAY and NVIDIA.

Figure 8 shows that Gemini-2.5's CPU gains are *environment* and *prompt* dependent. **On CRAY (left)**, all three prompts deliver substantial speedups, with medians around ~2.8–3.3×; *Context+Persona* achieves the highest observed speedup (up to ~3.5×). **On NVIDIA (CPU mode, right)**, performance is more variable: *Persona* and *Context* alone frequently underperform (medians < 1×), whereas *Context+Persona* is the only variant that consistently approaches or exceeds baseline, with median speedups in the ~0.9–1.3× range.

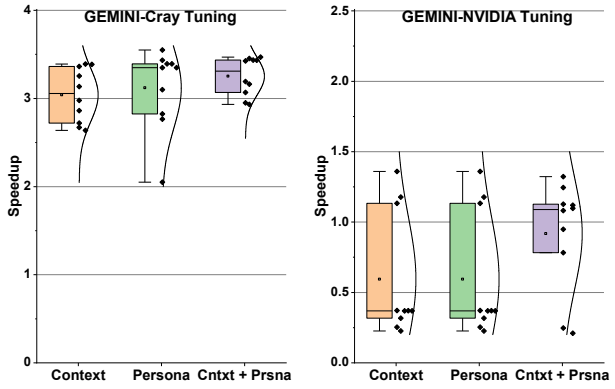


Figure 8: Gemini-2.5-based Tuning: CPU Speedup for the two programming environments: CRAY and NVIDIA.

Figure 9 shows a clear pattern for Claude 4.0: **on CRAY (left)**, the *Persona* pattern obtains the highest speedups (median ~5–5.5×, while *Context* and *Context+Persona* ranges from ~2.6–3.0×. **On NVIDIA (CPU mode, right)**, *Persona* is the only variant with a median near or slightly above baseline (~1.0–1.2×); *Context* and *Context+Persona* underperform (medians < 1×). For Claude-4.0, Persona prompting tends to produce loop orders with a better spatial locality, which the Cray compiler can exploit via auto-vectorization.

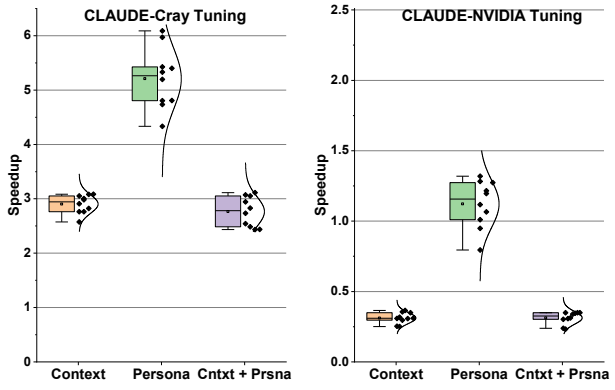


Figure 9: Claude4.0-based Tuning: CPU Speedup for the two programming environments: CRAY and NVIDIA.

3.2.2 ***GPU Speedup Evaluation.*** We now evaluated the performance of LoopGen-AI and ML-based autotuning (GPTune) on GPU execution. The kernels were compiled on both environments, **Cray** and **NVIDIA**. As in the CPU experiments, the graphs show the GPU speedup distribution during the course of auto-tuning.

Figure 10 (GPT-4.1 on GPU) shows a clear environment effect. **On CRAY–GPU (left)**, all three prompts deliver substantial gains, with medians in the ~2.2–3.0× range and best cases reaching ~3.1×. **On NVIDIA–GPU (right)**, medians are near baseline (~0.95–1.10×); among the patterns, *Context+Persona* obtains the highest speedup, up to ~1.33×.

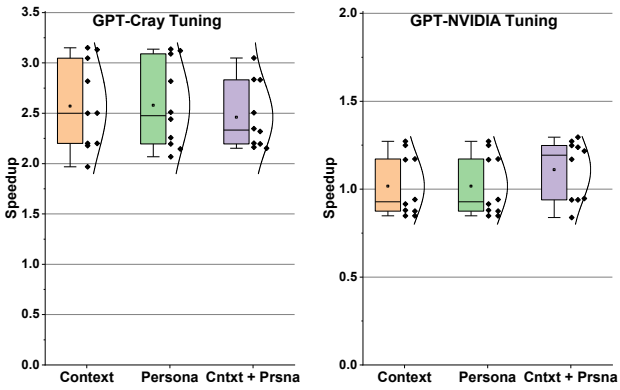


Figure 10: GPU GPT-4.1-based Tuning: Speedup for two of the programming environments: CRAY and NVIDIA.

Figure 11 (Gemini-2.5 on GPU) again shows a significant environment impact. **On CRAY–GPU (left)**, loop reordering delivers large gains: medians lie around ~2.6–3.3×, with *Context* and *Context+Persona* achieving the highest speedup, up to 3.3×. **On NVIDIA–GPU (right)**, *Context* and *Context+Persona* have medians near ~1.2–1.3× achieving speedups up to 1.3×, whereas *Persona* is more variable and includes occasional regressions (below 1×).

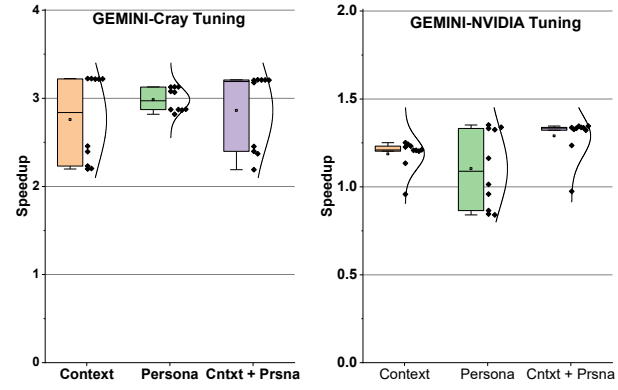


Figure 11: GPU Gemini-2.5-based Tuning: Speedup for two of the programming environments: CRAY and NVIDIA.

Figure 12 (Claude-4.0 on GPU) shows a pronounced prompt dependence. **On CRAY–GPU (left)**, *Context* and *Context+Persona* deliver the highest gains (medians ∼2.6–3.2×, obtaining a maximum speedup up to 3.2×), whereas *Persona* alone underperforms (median ∼1.0–1.2×). **On NVIDIA–GPU (right)**, *Context* yields a tight distribution near ∼1.1–1.2×; *Context+Persona* is slightly higher on average, obtaining a speedup up to 1.3×; *Persona* is the most variable and includes regressions (<1×).

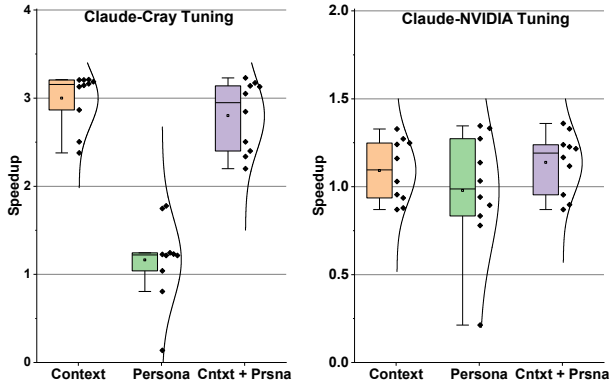


Figure 12: GPU Claude-4.0-based Tuning: Speedup for two of the programming environments: CRAY and NVIDIA.

4 Related work

Traditional Machine Learning autotuning approaches have demostrated success in exploring high-dimensional spaces [8] [4] [6] [22], but they often require many attempts of compile and run evaluations dealing with performance optimization as black box evaulated based on observed performance. In contrast, with the emergence of Agentic-AI systems, recent studies have evaluated the performance of autonomous agents to work on specific logical tasks, assigning various roles and responsibilities in multi-agent systems [7] [11] [17].

Another significant contribution on this domain is the incorporation of feedback obtained from the objective world or virtual environment. For instance, Voyager [18] incorporates three types of environment feedback; program execution, the execution error, and self verification results. The feedback from the environment help the agent to make better plans for its next action.

Other studies have incorporated varios LLMs to accomplish a wide range of task such as generating code. For instance, ChatDev [11], different agents discuss the software development process and take decisions on their own behavior.

To the best of our knowledge, no other study has compared an ML-based autotuner strategy with and Agentic AI System for code refactoring and optimization. Our study provides an evaluation methodology and several insights for future HPC performance engineering frameworks.

5 Conclusions

This paper addresses an existential question for the HPC community: **"How Agentic AI Systems Compare to Traditional ML Autotuning Techniques?"** To this end, we presented a comparative study between a ML-based autotuning (GPTune) and Agentic AI system (LoopGen-AI) for loop reordering optimization in a real-world case scenario. We evaluated the two approaches on the Real Time Dyson Expansion (RT-DE) kernel using the Perlmutter supercomputer at NERSC, targeting both AMD EPYC 7763 CPU and NVIDIA-A100 GPU, across two different programming environments: PrgEnv-CRAY and PrgEnv-NVIDIA.

Our main findings are that LoopGen-AI achieves competitive speedups with significantly fewer runs, while ML-based approaches, such as GPTune, require between 150-250 evaluations to identify the optimal loop reordering. The reasoning logs from the LLMs revealed that their decisions rely heavily on **the combination of semantic understanding of the kernel with dynamic feedback from the environment**, highlighting a promising new dimension in performance tuning. Additionally, our analysis of different **prompt patterns** showed that combining the **Persona** and **Context Manager** patterns leads to better loop reorderings. This emphasizes the importance of prompt engineering in Agentic AI systems.

Finally, addressing the main question asked in the introduction, we find that, with their current capabilities, the Agentic AI system is not yet a full replacement for ML-based autotuners, they offer compelling complementary strengths.

Future autotuning systems should aim to **integrate AI-driven semantic reasoning with statistical learning methods**, leveraging the best of both worlds. In future work, we plan to extend LoopGen-AI with multi-agent collaboration and explore its applicability to more diverse computational kernels.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program under Award Number DE-AC02-05CH11231, and used resources of the National Energy Research Scientific Computing Center (NERSC).

References

[1] Anthropic 2024. *Claude 4 Model.* Anthropic. https://www.anthropic.com/news/claude-4
[2] Google DeepMind 2024. *Gemini-2.5 Model.* Google DeepMind. https://deepmind.google/models/gemini/#technical-report
[3] OpenAI 2024. *GPT-4.1 website.* OpenAI. https://openai.com/index/gpt-4-1
[4] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation.* 303–316.
[5] Akshay Bhosale, Parinaz Barakhshan, Miguel Romero Rosas, and Rudolf Eigenmann. 2022. Automatic and Interactive Program Parallelization Using the Cetus Source to Source Compiler Infrastructure v2.0. *Electronics* 11, 5 (2022). doi:10.3390/electronics11050809
[6] Stefan Falkner, Aaron Klein, and Frank Hutter. 2018. BOHB: Robust and efficient hyperparameter optimization at scale. In *International conference on machine learning.* PMLR, 1437–1446.
[7] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. 2023. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations.*
[8] Yang Liu, Wissam M Sid-Lakhdar, Osni Marques, Xinran Zhu, Chang Meng, James W Demmel, and Xiaoye S Li. 2021. GPTune: Multitask learning for autotuning exascale applications. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* 234–246.

[9] NERSC Documentation Team. 2025. *NERSC Compilers and Wrappers*. https://docs.nersc.gov/development/compilers/wrappers

[10] Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*. 1–22.

[11] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. 2023. Chatdev: Communicative agents for software development. *arXiv preprint arXiv:2307.07924* (2023).

[12] Asif Rahman, Veljko Cvetkovic, Kathleen Reece, Aidan Walters, Yasir Hassan, Aneesh Tummeti, Bryan Torres, Denise Cooney, Margaret Ellis, and Dimitrios S Nikolopoulos. 2025. Marco: A multi-agent system for optimizing hpc code generation using large language models. *arXiv preprint arXiv:2505.03906* (2025).

[13] Cian C. Reeves, Michael Kurniawan, Yuanran Zhu, Nikil Jampana, Jacob Brown, Chao Yang, Khaled Z. Ibrahim, and Vojtech Vlcek. 2025. A Practical Framework for Simulating Time-Resolved Spectroscopy Based on a Real-Time Dyson Expansion. *Journal of Chemical Theory and Computation* 21, 14 (22 Jul 2025), 6667–6682. doi:10.1021/acs.jctc.5c00696

[14] Miguel Angel Romero Rosas, Miguel Angel Torres Sanchez, and Rudolf Eigenmann. 2025. Should AI Optimize Your Code? A Comparative Study of Classical Optimizing Compilers Versus Current Large Language Models. In *Proceedings of the 2025 Supercomputing Asia Conference (SCA '25)*. Association for Computing Machinery, New York, NY, USA, 22–29. doi:10.1145/3718350.3718357

[15] Jingqing Ruan, Yihong Chen, Bin Zhang, Zhiwei Xu, Tianpeng Bao, Hangyu Mao, Ziyue Li, Xingyu Zeng, Rui Zhao, et al. 2023. Tptu: Task planning and tool usage of large language model-based ai agents. In *NeurIPS 2023 Foundation Models for Decision Making Workshop*.

[16] Paul B Schneck. 1973. A survey of compiler optimization techniques. In *Proceedings of the ACM annual conference*. 106–113.

[17] Gregory Serapio-García, Mustafa Safdari, Clément Crepy, Luning Sun, Stephen Fitz, Marwa Abdulhai, Aleksandra Faust, and Maja Matarić. 2023. Personality traits in large language models. (2023).

[18] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An open-ended embodied agent with large language models, 2023. *URL https://arxiv. org/abs/2305.16291* (2023).

[19] 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.

[20] Zihao Wang, Shaofei Cai, Guanzhou Chen, Anji Liu, Xiaojian Ma, and Yitao Liang. 2023. Describe, explain, plan and select: Interactive planning with large language models enables open-world multi-task agents. *arXiv preprint arXiv:2302.01560* (2023).

[21] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382* (2023).

[22] Xingfu Wu, Prasanna Balaprakash, Michael Kruse, Jaehoon Koo, Brice Videau, Paul Hovland, Valerie Taylor, Brad Geltz, Siddhartha Jana, and Mary Hall. 2025. ytopt: Autotuning scientific applications for energy efficiency at large scales. *Concurrency and Computation: Practice and Experience* 37, 1 (2025), e8322.

[23] Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. 2024. Memorybank: Enhancing large language models with long-term memory. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 19724–19731.