Improving Parallel Program Performance with LLM Optimizers via Agent-System Interfaces

Anonymous Author(s)
Affiliation
Address
email

Abstract

Modern scientific discovery increasingly relies on high-performance computing for complex modeling and simulation. A key challenge in improving parallel program performance is efficiently mapping tasks to processors and data to memory, a process dictated by intricate, low-level system code known as *mappers*. Developing high-performance mappers demands days of manual tuning, posing a significant barrier for domain scientists without systems expertise. We introduce a framework that automates mapper development with generative optimization, leveraging richer feedback beyond scalar performance metrics. Our approach features the Agent-System Interface, which includes a Domain-Specific Language (DSL) to abstract away low-level complexity of system code and define a structured search space, as well as AutoGuide, a mechanism that interprets raw execution output into actionable feedback. Unlike traditional reinforcement learning methods such as OpenTuner, which rely solely on scalar feedback, our method finds superior mappers in far fewer iterations. With just 10 iterations, it outperforms OpenTuner even after 1000 iterations, achieving $3.8\times$ faster performance. Our approach finds mappers that surpass expert-written mappers by up to $1.34\times$ speedup across nine benchmarks while reducing tuning time from days to minutes.

1 Introduction

Modern scientific discovery depends on advanced software tools for modeling and simulation [1–3]. Computational scientists, including physicists, chemists, and biologists, rely on high-performance computing to tackle complex problems. These scientific computations dominate workloads on the world's most powerful supercomputers [4]. However, many domain scientists lack expertise in computer science, and therefore having difficulties in optimizing their programs because of the complexity and scale of the underlying machines. Even for experts, finding and fixing performance problems resulting from program modifications or when porting to a new machine is often time-consuming. Any progress on automating performance tuning is of great benefit in this domain.

Task-based programming [5–10] has emerged as a promising approach to high performance computing. The paradigm involves decomposing computations into independent *tasks* that communicate exclusively through their arguments. A key advantage of task-based systems is that the performance tuning problem is factored out into a separate *mapping*: an assignment of tasks to processors and data to particular memories. High-quality mapping, achieved through a well-designed *mapper* (implemented as code), can significantly improve performance, often by an order of magnitude [11].

However, currently writing mappers remains a labor-intensive process, as it requires deep knowledge of applications, hardware, and low-level system APIs. In addition, this process is highly application-specific, input-specific, and machine-specific, often taking experts several days of meticulous tuning

Submitted to 39th Conference on Neural Information Processing Systems (NeurIPS 2025). Do not distribute.

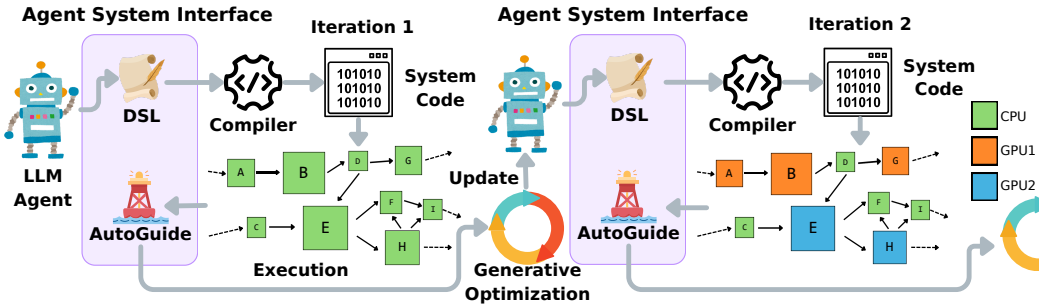


Figure 1: **Iterative mapper refinement with agent-based generative optimization.** The system leverages the Agent-System Interface, which consists of the Domain-Specific Language (DSL) and AutoGuide. The DSL abstracts away the low-level system code, defining a search space for mapping strategies, while AutoGuide interprets execution results into actionable guidance. As iterations progress, the mapper evolves to improve performance.

to achieve high performance. This challenge is especially pronounced for domain scientists, who typically lack the necessary expertise in computer systems and code optimization. Automating mapper development would enable scientists to focus on their own domain of expertise while fully utilizing the capabilities of high-performance computing systems.

In this paper, we introduce a system powered by large language models (LLMs) to **automate both the generation and optimization of mapper code**. The first challenge stems from the *complexity of generating mapper code* due to the original low-level programming system, which exposes the agent to intricate system APIs, coupled with the problem that raw feedback messages from the system are often uninformative to the agent. The second challenge involves *optimizing mapper performance*. Specifically, it consists of (1) defining an appropriate search space and (2) devising efficient methods to find optimal mappers, thereby maximizing parallel program performance.

To address the first challenge, we propose an **Agent-System Interface** (ASI), as shown in Figure 1, an abstraction layer between the agent and the system that simplifies code generation and provides more meaningful feedback to the agent. At the core of ASI is a *Domain-Specific Language (DSL)*, a high-level interface that encapsulates all performance-critical decisions required to generate a mapper. The DSL abstracts away the complexity of low-level system code with a compiler. Additionally, the DSL defines a structured search space, enabling systematic exploration of mapping strategies. We also design and implement the *AutoGuide* mechanism to interpret raw execution output into informative and actionable guidance. This mechanism allows the agent to iteratively optimize the mapper by leveraging enriched feedback to update its strategy.

For the second challenge, we adopt the **generative optimization** approach, a recent advance in optimization techniques. Unlike traditional methods such as reinforcement learning [12], which rely solely on scalar rewards, generative optimization can utilize richer forms of feedback, such as error explanations and actionable suggestions expressed in natural language. This agentic optimization workflow has previously proven to be effective across various domains [13–17]. Our work is the first to apply such technique to the domain of system optimization.

Our experiments demonstrate that mappers optimized by LLM-powered agents not only match but often surpass expert-written mappers, achieving up to $1.34\times$ speedup across nine benchmarks. Since expert-written mappers set the highest standard, surpassing them is a notable accomplishment. At the same time, our method significantly reduces mapper tuning time from days to minutes, making high-performance mapping more accessible to domain scientists. To further highlight the advantage of generative optimization, we compare it against OpenTuner, a reinforcement learning-based autotuning framework. Our generative optimizer finds mappers $11\times$ faster than OpenTuner when both run for 10 iterations and still maintains a $3.8\times$ advantage even when OpenTuner runs for 1000 iterations. Furthermore, ablation studies underscore the necessity of the agent-system interface design in achieving these performance gains. Our contributions are as follows:

1. **Design of an Agent-System Interface:** We introduce an abstraction layer that simplifies mapper code generation and provides guidance to the agent. The Domain-Specific Language (DSL) defines a search space, allowing the agent to explore mapping strategies without dealing with low-

level system code. AutoGuide interprets raw execution output into targeted feedback, enabling the agent to refine mapper code more effectively.

2. **Generative Optimization for Systems:** We introduce generative optimization to improve system performance, leveraging richer feedback such as error messages and actionable suggestions in natural language. Unlike reinforcement learning methods like OpenTuner, which rely solely on scalar feedback, our method identifies better mappers in far fewer iterations. With only 10 iterations, it outperforms OpenTuner by $3.8\times$ even after 1000 iterations.
3. **Empirical Evaluation of Performance:** Our agent-based solution achieves up to $1.34\times$ speedup across nine benchmarks, surpassing expert-written mappers while reducing tuning time from days to minutes. We highlight the critical role of the agent-system interface through ablation studies, demonstrating its impact on achieving the performance gains.

2 Related Work

Mapping in Parallel Programming Many parallel programming systems allow users to make their own mapping decisions, such as Legion [6], StarPU [7, 18], Chapel [8], HPX [19, 20], Sequoia [21], Ray [9], TaskFlow [22], and Pathways [10]. Several techniques have been proposed to automate mapping, including machine learning models [23, 24], static analysis [25, 26], reinforcement learning [12, 27] and auto-tuning [28]. We use an agent-based approach with LLMs and explore a larger search space for mappers than traditional methods.

Agentic Frameworks Agents powered by Large Language Models (LLMs) play a critical role in decision-making, planning, tool integration, and solving complex problems in dynamic environments [29]. Many agentic frameworks have been developed [30–33], with uses spanning domains such as software engineering [34–36], robotics [37], healthcare [38], education [39], and knowledge engineering [40]. Our work is the first to apply an agentic workflow to iteratively optimize mapper code, improving the performance of parallel programs.

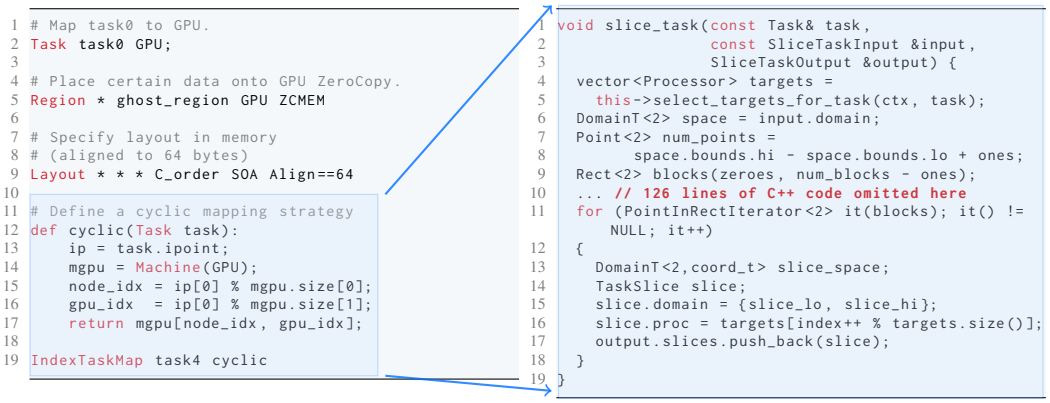
AI for Systems The application of AI to optimize system design has gained significant traction in recent years. Techniques such as deep learning [41–43] and gradient-boosted trees [44] have been used to predict program execution times for performance optimization. Reinforcement learning methods have addressed challenges in chip floorplanning [45], autotuning [12], auto-vectorization [46], and compiler phase ordering [47]. While previous efforts have predominantly relied on traditional approaches for cost prediction and optimization, our work uses the recent advances in generative optimization to tackle complex system challenges.

Generative Optimization Recent work has explored the use of LLMs for optimization problems traditionally tackled with numerical methods, including mixed-integer programming [48, 49] and numerical optimization [13]. A key advantage of generative optimization is its ability to iteratively refine solutions using diverse forms of feedback. For example, Cheng et al. [14] applies generative optimization to robotic manipulation and game playing, while Yuksekgonul et al. [17] optimizes prompts and molecular designs. While reinforcement learning has been applied to system optimization, the potential of LLM-driven optimization in systems remains unexplored. Our work explores whether generative optimization with richer feedback outperforms traditional methods using scalar rewards in system optimization.

3 Problem Definition

Motivation and Challenges The concrete problem we address is the automated generation of high-performance mappers for the Legion parallel programming framework [6]. Mappers dictate task scheduling and data placement. A well-designed mapper can achieve orders-of-magnitude speedup over naive strategies.

However, automating mapper generation is challenging due to two key factors. First, **the complexity of low-level system code**. Implementing a mapper requires writing hundreds of lines of intricate C++ code, demanding expertise in system internals. Second, **the vast search space of mapping strategies**. The search space grows exponentially with the number of tasks and arguments.



```
# Map task0 to GPU.
Task task0 GPU;

# Place certain data onto GPU ZeroCopy.
Region * ghost_region GPU ZCMEM

# Specify layout in memory
# (aligned to 64 bytes)
Layout * * * C_order SOA Align==64

# Define a cyclic mapping strategy
def cyclic(Task task):
    ip = task.ipoint;
    mgpu = Machine(GPU);
    node_idx = ip[0] % mgpu.size[0];
    gpu_idx  = ip[0] % mgpu.size[1];
    return mgpu[node_idx, gpu_idx];

IndexTaskMap task4 cyclic
```

(a) An example mapper in Domain-Specific Language (DSL)

```
void slice_task(const Task& task,
                const SliceTaskInput &input,
                SliceTaskOutput &output) {
  vector<Processor> targets =
    this->select_targets_for_task(ctx, task);
  DomainT<2> space = input.domain;
  Point<2> num_points =
        space.bounds.hi - space.bounds.lo + ones;
  Rect<2> blocks(zeroes, num_blocks - ones);
  ... // 126 lines of C++ code omitted here
  for (PointInRectIterator<2> it(blocks); it() !=
      NULL; it++)
  {
    DomainT<2,coord_t> slice_space;
    TaskSlice slice;
    slice.domain = {slice_lo, slice_hi};
    slice.proc = targets[index++ % targets.size()];
    output.slices.push_back(slice);
  }
}
```

(b) Code snippet from a C++ mapper

Figure 2: **Comparison of a DSL mapper and a C++ mapper.** The DSL's **declarative, high-level** design abstracts away the complexity of low-level C++ code, serving as the core of the **Agent-System Interface**. The highlighted boxes illustrate how the same functionality, which requires extensive C++ system code, can be expressed concisely in just a few lines in DSL.

Search Space and Performance Impact As illustrated in Figure A1, the search space of mappers involves multiple decisions, each influencing performance. The first key aspect is **processor selection**, which determines whether a task runs on GPUs, CPUs, or the OpenMP runtime. This choice depends on factors such as task size, GPU memory capacity, and kernel launch overhead. For instance, small tasks may prefer CPUs due to the overhead of launching GPU kernels, while tasks with large memory footprints may run on CPUs when GPU memory is insufficient.

Another crucial dimension is **memory placement**, which dictates where data is stored. A mapper must decide whether to place data in the GPU's FrameBuffer for fast access, ZeroCopy memory for CPU-GPU sharing, or CPU system memory for more available storage. Each option presents trade-offs between access speed, memory usage, and data transfer overhead.

Additionally, **memory layout** further expands the search space, with decisions on Struct of Arrays (SOA) vs. Array of Structures (AOS), data ordering (Fortran-order vs. C-order), and alignment constraints (e.g., 128-byte alignment) significantly affecting cache efficiency and performance.

Finally, an important idiom in high-performance computing is launching tasks over partitioned data. **Index mapping** determines how data partitions and task executions are distributed across multiple processors. For consistency, we can represent data partitioning as a tensor of data partitions, the machine as a tensor of processors, and tasks operating on the partitioned data as a tensor of tasks. The way data and task indices are mapped to processor indices affects inter-processor communication, a key factor in performance [50, 51].

4 Our Approach: Agent-System Interface

4.1 Domain-Specific Language Design

A key challenge in automating mapper generation with a coding agent is the complexity of low-level system code, which requires intricate C++ implementations. To address this, we design a high-level **Domain-Specific Language (DSL)** as the core of our **Agent-System Interface** (ASI). The DSL provides a *structured search space* for mapping strategies while *abstracting away low-level implementation details*. Unlike C++, which demands imperative specifications of mapping policies, our DSL adopts a **declarative design**, allowing users to specify *what* to achieve rather than *how* to implement it. Most critically, the DSL **separates concerns**, enabling multiple aspects of mapping decisions to be expressed *independently rather than being entangled* in low-level system APIs. This design reduces code complexity and naturally provides a search space for the agent to explore. To implement it, we develop a *compiler* that translates DSL into the low-level C++ APIs.

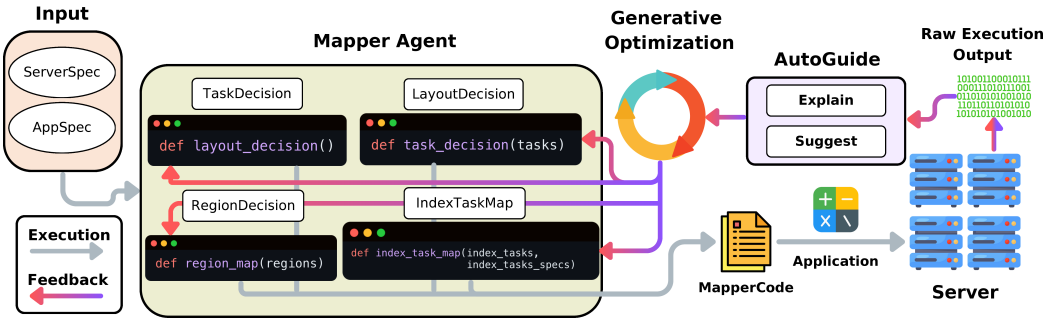


Figure 3: **Agent Optimization Process**. The mapper agent takes server specifications and application-specific information as input, generates mapper code, and executes it alongside the application on the server. Raw execution feedback is enriched using the **AutoGuide** mechanism and iteratively refined by an LLM optimizer to improve performance.

As illustrated in Figure 2, the complexity of DSL code is significantly lower than that of C++. Figure 2a provides an example of a DSL mapper, highlighting the key features of our DSL. In contrast, Figure 2b shows a snippet from a C++ mapper, emphasizing the intricacy of low-level implementation details. According to Table A1, using the DSL results in an **average lines of code reduction of** $14\times$. This substantial reduction makes DSL a more suitable target for LLM code generation, as it abstracts away the complexities inherent in low-level systems. As we will show in Section 5.2, LLMs generate DSL code more effectively, despite DSL having no examples in LLM training corpora, whereas C++ is widely represented.

Next, we describe the DSL's design, emphasizing its declarative nature and structured search space. Section 3 details the performance impact of each decision.

The `Task` statement (Line 2) defines processor selection for each task, choosing between CPU, GPU, or OpenMP. Line 2 specifies that instances of `task0` should run on GPUs. This decision is made per task; note that the search space expands exponentially with the number of tasks.

The `Region` statement (Line 5) controls memory placement for data arguments. Line 5 specifies that all tasks using `ghost_region` should place the data in GPU ZeroCopy memory. Other choices include GPU FrameBuffer memory and CPU System Memory. This decision is made per task and per argument, causing the search space to grow exponentially.

The `Layout` statement (Line 9) defines memory layouts. Line 9 enforces a `C_order` axis ordering, an `SOA` layout, and a 64-byte memory alignment for all data used by all tasks mapped to all processors. Alternative choices include `F_order`, `AOS`, and various alignment strategies. This is a per-task, per-data, per-processor decision.

The `IndexTaskMap` statement (Line 19) controls index mapping using a customized function. Line 12 defines the mapping function that establishes the correspondence between two index spaces: the task index space (represented by `task.ipoint`) defined in the application code (e.g., for loops) and the processor space of the distributed machine (represented by `Machine(GPU)`). The DSL allows users to express arbitrary arithmetic mappings between the two index spaces. This decision applies to each task group launched by parallel for loops.

4.2 Generative Optimization via AutoGuide

We formulate mapper generation as an **online optimization problem**. Given a triplet $(\Theta, \omega, \mathcal{T})$, where Θ is a set of possible mappers, ω is an *optimization objective*, and $\mathcal{T}$ is a function that takes a mapper $\theta \in \Theta$ as input, $(f, g) = \mathcal{T}(\theta)$ and returns f, the *feedback* from executing the mapper (i.e., the measured performance after running the application code with the generated mapper), and g, the *process graph* tracing how the mapper was generated. In our setup, mapper performance is deterministic, as we carefully control all sources of randomness in the environment. If the parameter space were numerical, this online optimization problem could be addressed using bandit algorithms [52], reinforcement learning [53], or Bayesian optimization [54], but these methods are less efficient when the parameter search space is large and discrete (i.e., text).

Case	Raw Execution Output	AutoGuide	
		Explain	Suggest
Case 1	**Execution Error:** Assertion failed: stride does not match expected value.	Memory layout is unexpected.	Adjust the layout constraints or move tasks to different processor types.
Case 2	**Performance Metric:** Execution time is 0.03s.	N/A	Move more tasks to GPU to reduce execution time.

Table 1: **AutoGuide Feedback Mechanism.** The AutoGuide mechanism interprets raw execution output from the runtime system, providing more informative error explanations and suggestions for mapper modifications. It is implemented via keyword matching. Additional examples are shown in Table A2.

In this online optimization problem, we leverage the DSL to structure the parameter space to improve the efficiency of optimization. Here, θ represents the program code, while ω and f are expressed as text. We adopt **generative optimization**, leveraging LLMs as optimizers given the objective in text form. This emergent optimization behavior has been recently observed and applied across various domains [55, 14, 17, 56].

Optimization Process We present the optimization process in Figure 3. The agent takes two inputs: server specifications (e.g., CPU/GPU counts) and application information (e.g., task lists, data arguments). It generates mapper code, which is executed alongside the application code on the server. Raw execution feedback from the runtime is augmented with the AutoGuide mechanism and fed back to the LLM, iteratively refining the agent for improved mapper code generation.

Coding Agent Our mapper agent improves mapping decisions by iteratively generating DSL code. A high-level schema of the mapper agent is shown in Figure 3. The mapper agent is implemented as a Python program in the Trace [14] framework, where we decompose the task of generating a monolithic mapper into *independent code segments*. This decomposition allows the agent to decide what code to generate for each segment separately. This approach is effective because our *DSL design eliminates unnecessary dependencies* between mapping decisions. Our modularization strategy aligns with least-to-most prompting [57].

AutoGuide The AutoGuide feedback mechanism is designed based on three key motivations: (1) generative optimization benefits from natural language feedback rather than relying solely on scalar values, (2) raw execution output from the runtime system is often too uninformative to effectively guide the agent's decisions, and (3) domain heuristics known to systems researchers can be naturally expressed in language (e.g., most tasks run faster on GPUs than CPUs). To address these needs, AutoGuide helps the agent by **explaining** opaque error messages and **suggesting** mapper modifications. As shown in Table 1, it interprets uninformative execution output into actionable insights, with additional examples in Appendix A.4. The implementation relies on keyword matching over the raw execution output. An ablation study in Section 5.3 demonstrates its effectiveness in our experiments.

5 Evaluation

Experiments are conducted on one node with two Intel 10-core E5-2640 v4 CPUs, 256G main memory, and four NVIDIA Tesla P100 GPUs. We use gpt-4o-2024-08-06.

5.1 Speedup of Application Performance

We evaluate our approach using 9 benchmarks. Circuit [6] is a simulation benchmark that models electrical circuit behavior by simulating currents and voltages across interconnected nodes and wires. Stencil [58] simulates a 2D grid where each point's value is updated based on a stencil pattern determined by its neighbors. Pennant [59] models unstructured mesh Lagrangian staggered-grid hydrodynamics, commonly used for simulating compressible flow. The remaining six benchmarks – Cannon's [60], SUMMA [61], PUMMA [62], Johnson's [63], Solomonik's [64], and COSMA [65]

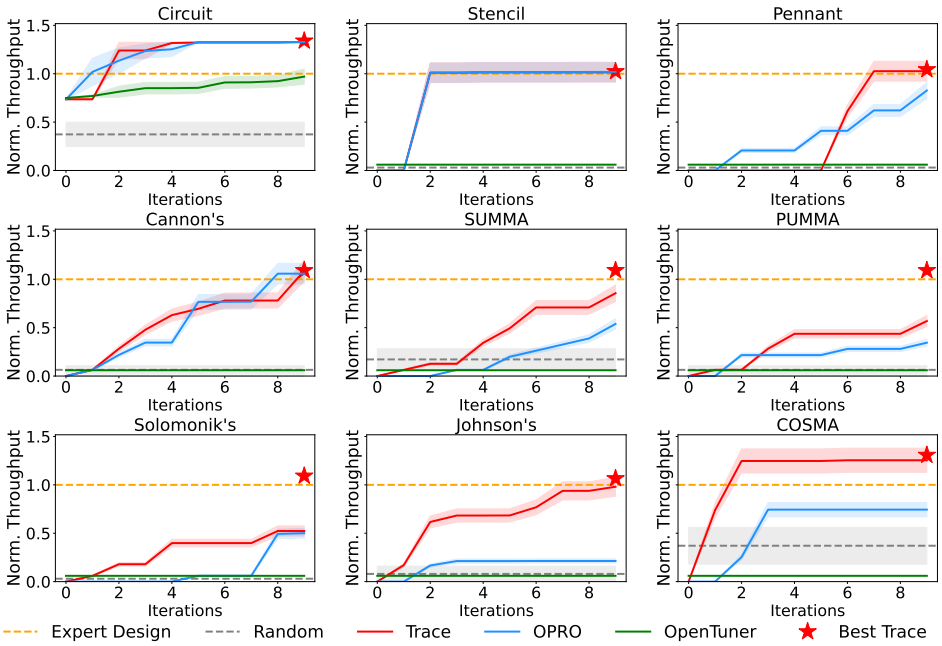


Figure 4: **Performance Comparison.** Normalized throughput for 9 benchmarks, comparing expert mappers, random mappers, the average optimization trajectories of Trace, OPRO, and OpenTuner in 10 iterations across 5 runs, and the best mappers found by Trace.

– are widely studied parallel matrix multiplication algorithms, which we discuss in more detail in Appendix A.3.

In this experiment, we evaluate the performance of the mappers with the following baselines.

Expert-Written Mappers. These mappers are manually developed by domain scientists who spend years mastering computational science. Writing mappers in parallel programming frameworks is another challenge, and tuning them for specific applications can take days.

Randomly Generated Mappers. These mappers were randomly generated with 10 different random seeds, sampling from the entire search space of each application. We report the average performance.

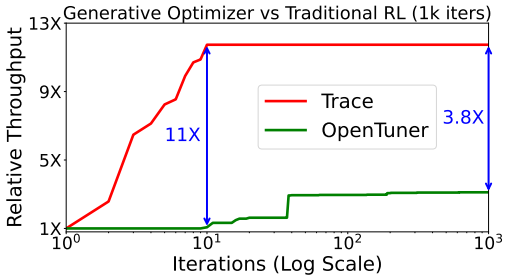


Figure 5: Comparison of Trace (generative optimizer) and OpenTuner (traditional RL) over 1K iterations (averaged across all 9 benchmarks).

Agent-Optimized Mappers. Using Trace [14], we evaluated the **Trace** and **OPRO** [15] search algorithms, running 10 iterations per application. To account for stochastic output, we repeated the process 5 times and report the average. The best mapper from Trace across runs is also reported.

OpenTuner Mappers. OpenTuner [12] is a program autotuning framework that uses reinforcement learning to optimize performance based on scalar feedback. We provided execution time as feedback, with a high penalty for failures.

Results We use normalized throughput as the performance metric in Figure 4, where higher values indicate better performance. The throughput is normalized relative to the expert-written mappers. **All the best mappers found by Trace can match or surpass the expert-written mappers**, underscoring the effectiveness of agent-based generative optimizer. Random mappers consistently exhibit low performance across all applications, emphasizing the critical role of mapping decisions. When

Code Generation Target	Mapping Strategy 1	2	3	4	5	6	7	8	9	10	Success Rate
C++ (single trial)	✗	–	–	✗	–	–	✗	✗	–	–	0%
DSL (single trial)	✓	✓	✓	✓	✓	–	✓	✓	✓	–	80%
C++ (iterative refine)	✗	–	–	✗	✗	✗	✗	✗	✗	✗	0%
DSL (iterative refine)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	100%

Table 2: **Code Generation Success Rates.** Success rates for generating code across 10 mapping strategies described in natural language. The test evaluates whether the generated code compiles and passes execution tests. Generating DSL code significantly outperforms generating C++ for both settings. Symbols indicate results: – fails to compile, ✗ compiles but fails the test, and ✓ passes the test.

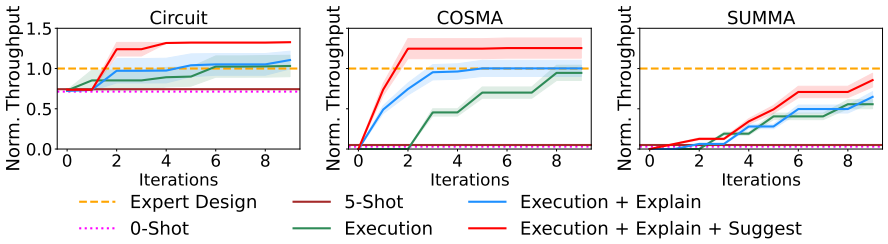


Figure 6: **Comparison of different feedback designs. 0-Shot** and **5-Shot** are baselines. **Execution** provides only the raw execution output as feedback. **Explain** provides additional explanations of execution errors. **Suggest** offers mapper modification suggestions. All feedback is automatically generated.

comparing optimization trajectories, Trace performs similarly to OPRO, and significantly outperforms OpenTuner.

To further compare the agent-based optimizer with traditional reinforcement learning, we extended OpenTuner's optimization iterations from 10 to 1000, as shown in Figure 5, where the x-axis is the log-scale of iterations and the y-axis represents relative throughput (averaged across all 9 benchmarks). Notably, Trace achieves a $3.8\times$ speedup over OpenTuner even when OpenTuner is run for 1000 iterations. When both are limited to 10 iterations, Trace outperforms OpenTuner by $11\times$, demonstrating its ability to quickly identify high-performance mappings. **This highlights the superiority of Trace (generative optimizer) over OpenTuner (traditional reinforcement learning).** Moreover, Trace completes the entire optimization process in just 10 minutes per application, **reducing mapper development time from days to minutes**.

Case Analysis The largest performance gain achieved by Trace over the expert mapper is observed in Circuit, with a speedup of $1.34\times$. This improvement is primarily due to *memory placement*: the best mapper allocates two data collections to GPU FrameBuffer memory, while the expert mapper places them in GPU ZeroCopy memory. Despite a slight increase in inter-GPU communication costs, Trace reduces task execution time due to faster memory access, resulting in higher overall performance. For matrix-multiplication algorithms, the greatest speedup is seen in COSMA, with Trace achieving a $1.31\times$ speedup over the expert mapper. This is attributed to Trace's more efficient index mapping functions, which *reduce inter-GPU communication* by better distributing partitioned submatrices across GPUs. For additional context, examples of Trace mappers are presented in Appendix A.7.

5.2 Ablation Study of DSL for Code Generation

In Section 5.1, we demonstrate the overall effectiveness of our approach. Here, we conduct an ablation study on the DSL, the core of the Agent-System Interface. Since successful generation is the foundation of optimization, this subsection focuses on **how well the DSL helps LLMs *generate* correct mappers compared to C++**, rather than directly *optimizing* performance.

Experiment Setup We designed 10 mapping strategies, described in natural language, to evaluate whether LLMs can generate correct code in both the DSL and the original low-level C++. The strategies are detailed in Appendix A.6. To ensure a fair comparison, identical prompt materials (documentation, examples, and starting code) were provided for both the DSL and C++. Success rates are measured based on whether the generated code passes predefined test cases, with results reported for single trials and iterative refinement, where the LLM is allowed up to 10 iterations to improve the code using compiler feedback. The evaluation is conducted with the DSPy [16] framework.

Results Table 2 shows that **DSL achieves significantly higher generation success rates than C++** in both the single-trial and iterative refinement settings. This demonstrates the effectiveness of DSL's design in abstracting system complexity and providing a high-level interface that enables LLMs to tackle complex system challenges in code generation. Incorporating iterative refinement with compiler feedback further improves success rates, resolving three compilation errors in C++ and two in the DSL. However, the gap between DSL mappers and C++ mappers remains substantial. Notably, these results are striking given that the DSL is a low-resource language with no pre-training or fine-tuning data, while C++ code is widely present in LLM training corpora.

Analysis LLMs perform better with the DSL for two reasons. First, the **semantic gap** between natural language and code is smaller with the DSL than with C++. For example, writing a mapper to "align all data to 64 bytes in memory and use Fortran ordering" requires one line `Layout * * * Align==64 F_order` in the DSL because of its *declarative design*. In contrast, the C++ mapping API requires a sequence of operations to enforce alignment and ordering, which widens the semantic gap. Second, the DSL reduces the **amount of code**. As shown in Table A1, LLMs achieve an average reduction of $14\times$ in lines of code, simplifying code generation. These results underscore the importance of a high-level agent-system interface.

5.3 Ablation Study of the AutoGuide Feedback

The AutoGuide mechanism provides enriched feedback to the agentic optimizer. We compare with alternative feedback designs.

Experiment Setup We compare the following baselines. **0-shot** and **5-shot** have no feedback, allowing the LLM to generate once with either 0 or 5 examples provided. **Execution** only provides raw execution feedback, **Explain** offers additional explanations for execution errors, and **Suggest** offers mapper modification suggestions. The Trace trajectory shown in Figure 4 uses the full AutoGuide mode with all **Execution+Explain+Suggest**. As an ablation study, we evaluate 3 benchmarks.

Results and Analysis Figure 6 demonstrates that the **full feedback mechanism consistently outperforms** all reduced feedback variants. The 0-shot and 5-shot results perform the worst, underscoring the importance of feedback-based iterative refinement. This highlights the value of an agentic workflow, showing that performance improvements are not solely driven by prompting the LLM but are a direct result of the iterative refinement inherent in the workflow design.

6 Conclusion

In this paper, we introduced a system that leverages LLMs to automate mapper generation and optimization. The Agent-System Interface (ASI) simplifies code generation with a Domain-Specific Language (DSL), which abstracts away the low-level complexity of system code, and enriches execution feedback through AutoGuide, which interprets raw execution output into actionable guidance. We adopted generative optimization, allowing LLMs to refine mappers using rich textual feedback beyond scalar metrics. Unlike RL-based methods like OpenTuner, which rely on numerical rewards, our approach incorporates error explanations and targeted suggestions, accelerating search efficiency. Experiments show that agent-generated mappers outperform expert-written ones, achieving up to $1.34\times$ speedup across nine benchmarks. Our method, running only 10 iterations, maintains a $3.8\times$ advantage over OpenTuner even after 1000 iterations. By reducing mapper development time from days to minutes, our approach benefits computational scientists and demonstrates the effectiveness of generative optimization in system design.

References

[1] Ryan Stocks, Jorge L Galvez Vallejo, CY Fiona, Calum Snowdon, Elise Palethorpe, Jakub Kurzak, Dmytro Bykov, and Giuseppe MJ Barca. Breaking the million-electron and 1 eflop/s barriers: Biomolecular-scale ab initio molecular dynamics using mp2 potentials. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2024.

[2] Xiao Wang, Siyan Liu, Aristeidis Tsaris, Jong-Youl Choi, Ashwin M Aji, Ming Fan, Wei Zhang, Junqi Yin, Moetasim Ashfaq, Dan Lu, et al. Orbit: Oak ridge base foundation model for earth system predictability. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2024.

[3] Hatem Ltaief, Rabab Alomairy, Qinglei Cao, Jie Ren, Lotfi Slim, Thorsten Kurth, Benedikt Dorschner, Salim Bougouffa, Rached Abdelkhalak, and David E Keyes. Toward capturing genetic epistasis from multivariate genome-wide association studies using mixed-precision kernel ridge regression. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2024.

[4] Exascale Computing. DOE Explains Exascale Computing, 2025. https://www.energy.gov/science/doe-explainsexascale-computing.

[5] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. Regent: A high-productivity programming language for hpc with logical regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[6] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012.

[7] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. In *European Conference on Parallel Processing*, pages 863–874. Springer, 2009.

[8] Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, 21 (3):291–312, 2007.

[9] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, 2018.

[10] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Daniel Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, et al. Pathways: Asynchronous distributed dataflow for ml. *Proceedings of Machine Learning and Systems*, 4:430–449, 2022.

[11] Juan J Galvez, Nikhil Jain, and Laxmikant V Kale. Automatic topology mapping of diverse large-scale parallel applications. In *Proceedings of the International Conference on Supercomputing*, pages 1–10, 2017.

[12] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 303–316, 2014.

[13] Allen Nie, Ching-An Cheng, Andrey Kolobov, and Adith Swaminathan. The importance of directional feedback for llm-based optimizers. *arXiv preprint arXiv:2405.16434*, 2024.

[14] Ching-An Cheng, Allen Nie, and Adith Swaminathan. Trace is the next autodiff: Generative optimization with rich feedback, execution traces, and llms. *arXiv preprint arXiv:2406.16218*, 2024.

[15] Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. arxiv 2023. *arXiv preprint arXiv:2309.03409*.

[16] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, et al. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023.

[17] Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. Textgrad: Automatic" differentiation" via text. *arXiv preprint arXiv:2406.07496*, 2024.

[18] Cédric Augonnet, Jérôme Clet-Ortega, Samuel Thibault, and Raymond Namyst. Data-Aware Task Scheduling on Multi-Accelerator based Platforms. In *16th International Conference on Parallel and Distributed Systems*, pages 291–298, Shangai, China, December 2010. IEEE. URL https://hal.inria.fr/inria-00523937.

[19] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, pages 1–11, 2014.

[20] Thomas Heller, Patrick Diehl, Zachary Byerly, John Biddiscombe, and Hartmut Kaiser. Hpx–an open source c++ standard library for parallelism and concurrency. *Proceedings of OpenSuCo*, 5, 2017.

[21] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the memory hierarchy. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, volume 0 of *SC '06*, page 83–es, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 0769527000.

[22] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. Taskflow: A lightweight parallel and heterogeneous task graph computing system. *IEEE Transactions on Parallel and Distributed Systems*, 33(6):1303–1320, 2021.

[23] Michael F. P. O'Boyle, Zheng Wang, and Dominik Grewe. Portable mapping of data parallel programs to opencl for heterogeneous systems. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, page 1–10, USA, 2013. IEEE Computer Society. ISBN 9781467355247. doi: 10.1109/CGO.2013.6494993. URL https://doi.org/10.1109/CGO.2013.6494993.

[24] Zheng Wang and Michael F.P. O'Boyle. Mapping parallelism to multi-cores: A machine learning based approach. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, page 75–84, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583976. doi: 10.1145/1504176.1504189. URL https://doi.org/10.1145/1504176.1504189.

[25] Gabriel Poesia, Breno Guimarães, Fabrício Ferracioli, and Fernando Magno Quintão Pereira. Static placement of computation on heterogeneous devices. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA), October 2017.

[26] Manman Ren, Ji Young Park, Mike Houston, Alex Aiken, and William J. Dally. A tuning framework for software-managed memory hierarchies. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, page 280–291, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605582825. doi: 10.1145/1454115.1454155. URL https://doi.org/10.1145/1454115.1454155.

[27] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *International conference on machine learning*, pages 2430–2439. PMLR, 2017.

[28] Thiago SFX Teixeira, Alexandra Henzinger, Rohan Yadav, and Alex Aiken. Automated mapping of task-based programs onto distributed and heterogeneous machines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2023.

[29] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V Chawla, Olaf Wiest, and Xiangliang Zhang. Large language model based multi-agents: A survey of progress and challenges. *arXiv preprint arXiv:2402.01680*, 2024.

[30] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

[31] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*, 2023.

[32] Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in Neural Information Processing Systems*, 36:51991–52008, 2023.

[33] Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

[34] Izzeddin Gur, Hiroki Furuta, Austin Huang, Mustafa Safdari, Yutaka Matsuo, Douglas Eck, and Aleksandra Faust. A real-world webagent with planning, long context understanding, and program synthesis. *arXiv preprint arXiv:2307.12856*, 2023.

[35] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024.

[36] Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. From llms to llm-based agents for software engineering: A survey of current, challenges and future. *arXiv preprint arXiv:2408.02479*, 2024.

[37] Shyam Sundar Kannan, Vishnunandan LN Venkatesh, and Byung-Cheol Min. Smart-llm: Smart multi-agent robot task planning using large language models. In *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 12140–12147. IEEE, 2024.

[38] Junkai Li, Siyu Wang, Meng Zhang, Weitao Li, Yunghwei Lai, Xinhui Kang, Weizhi Ma, and Yang Liu. Agent hospital: A simulacrum of hospital with evolvable medical agents. *arXiv preprint arXiv:2405.02957*, 2024.

[39] Elkin Arturo Betancourt Ramirez and Juan Antonio Fuentes Esparrell. Artificial intelligence (ai) in education: Unlocking the perfect synergy for learning. *Educational Process: International Journal*, 13(1):35–51, 2024.

[40] Yijia Shao, Yucheng Jiang, Theodore A Kanell, Peter Xu, Omar Khattab, and Monica S Lam. Assisting in writing wikipedia-like articles from scratch with large language models. *arXiv preprint arXiv:2402.14207*, 2024.

[41] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pages 863–879, 2020.

[42] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 859–873, 2020.

[43] Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. Amos: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 874–887, 2022.

[44] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, et al. Tensorir: An abstraction for automatic tensorized program optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 804–817, 2023.

[45] Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Wenjie Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Azade Nazi, et al. A graph placement methodology for fast chip design. *Nature*, 594(7862):207–212, 2021.

[46] Ameer Haj-Ali, Nesreen K Ahmed, Ted Willke, Yakun Sophia Shao, Krste Asanovic, and Ion Stoica. Neurovectorizer: End-to-end vectorization with deep reinforcement learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, pages 242–255, 2020.

[47] Ameer Haj-Ali, Qijing Jenny Huang, John Xiang, William Moses, Krste Asanovic, John Wawrzynek, and Ion Stoica. Autophase: Juggling hls phase orderings in random forests with deep reinforcement learning. *Proceedings of Machine Learning and Systems*, 2:70–81, 2020.

[48] Ali AhmadiTeshnizi, Wenzhi Gao, Herman Brunborg, Shayan Talaei, and Madeleine Udell. OptiMUS-0.3: Using large language models to model and solve optimization problems at scale. *Submitted*, 2024. URL https://arxiv.org/abs/2407.19633.

[49] Ali AhmadiTeshnizi, Wenzhi Gao, and Madeleine Udell. OptiMUS: Scalable optimization modeling using MIP solvers and large language models. In *International Conference on Machine Learning (ICML)*, 2024. URL https://arxiv.org/abs/2402.10172.

[50] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, et al. Unity: Accelerating {DNN} training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 267–284, 2022.

[51] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022.

[52] Tor Lattimore and Csaba Szepesvári. *Bandit algorithms*. Cambridge University Press, 2020.

[53] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[54] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, 25, 2012.

[55] Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *International Conference on Learning Representations*, 2024.

[56] Bhrij Patel, Souradip Chakraborty, Wesley A Suttle, Mengdi Wang, Amrit Singh Bedi, and Dinesh Manocha. Aime: Ai system optimization via multiple llm evaluators. *arXiv preprint arXiv:2410.03131*, 2024.

[57] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, et al. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*, 2022.

[58] Rob F Van der Wijngaart and Timothy G Mattson. The parallel research kernels. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2014.

[59] Charles R Ferenbaugh. Pennant: an unstructured mesh mini-app for advanced architecture research. *Concurrency and Computation: Practice and Experience*, 27(17):4555–4572, 2015.

[60] Lynn Elliot Cannon. *A cellular computer to implement the Kalman filter algorithm*. Montana State University, 1969.

[61] Robert A Van De Geijn and Jerrell Watts. Summa: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[62] Jaeyoung Choi, David W Walker, and Jack J Dongarra. Pumma: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers. *Concurrency: Practice and Experience*, 6(7):543–570, 1994.

[63] Ramesh C Agarwal, Susanne M Balle, Fred G Gustavson, Mahesh Joshi, and Prasad Palkar. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development*, 39(5):575–582, 1995.

[64] Edgar Solomonik and James Demmel. Communication-optimal parallel 2.5 d matrix multiplication and lu factorization algorithms. In *Euro-Par 2011 Parallel Processing: 17th International Conference, Euro-Par 2011, Bordeaux, France, August 29-September 2, 2011, Proceedings, Part II 17*, pages 90–109. Springer, 2011.

[65] Grzegorz Kwasniewski, Marko Kabić, Maciej Besta, Joost VandeVondele, Raffaele Solcà, and Torsten Hoefler. Red-blue pebbling revisited: near optimal parallel matrix-matrix multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–22, 2019.

A Appendix

A.1 Illustration for Mapping

We show an illustration for mapping in Figure A1.

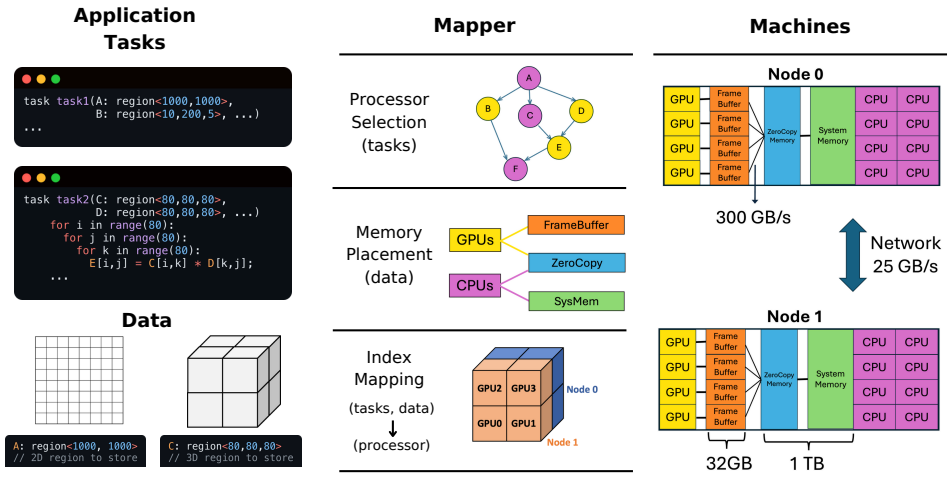


Figure A1: Mappers decide the placement of each task in the task graph to processors, the placement of data to memory, and how the iteration space of data is partitioned and mapped to different processors.

A.2 Tables for Lines of Code Comparison

We show the lines of code comparison between DSL and C++ mappers in Table A1.

Application	1	2	3	4	5	6	7	8	9	Avg.
LoC in C++	347	306	379	447	437	430	428	433	448	406
LoC in DSL	16	14	16	38	38	38	33	38	32	29
LoC Reduction	22×	22×	24×	12×	12×	11×	13×	11×	14×	14×

Table A1: **Lines of Code (LoC) comparison between DSL and C++ mappers**. The DSL achieves a 14× average reduction in LoC, making it a more suitable target for LLM code generation.

A.3 Parallel Matrix Multiplication Algorithms

2D Algorithms Cannon's [60] introduced a systolic communication pattern with tiled data partitioning for distributed matrix multiplication. PUMMA [62] and SUMMA [61] extended this approach by supporting non-square matrices and improving communication efficiency through pipelining. They are called 2D algorithms because they partition the matrices into 2D tiles and then map them onto the processor space.

Non-2D Algorithms Johnson's [63] introduced a 3D algorithm that partitions the input matrices into 3D tiles and uses additional memory per processor to reduce communication compared to 2D algorithms. Solomonik's [64] balances between 2D and 3D approaches by using extra memory to further minimize communication. COSMA [65] takes a different approach by optimizing the processor grid and parallelization strategy based on the input size and the machine size.

A.4 Examples of Feedback Configurations

We give examples for the raw execution output and enriched feedback in Table A2. The enhanced feedback includes explanations of errors and suggestions for mapper modifications.

Case	Raw Execution Output	AutoGuide	
		Explain	Suggest
`case1`	**Compile Error:** Syntax error, unexpected :, expecting {	N/A	There should be no colon : in function definition.
`case2`	**Compile Error:** IndexTaskMap's function undefined	N/A	Define the IndexTaskMap function first before using it.
`case3`	**Compile Error:** mgpu not found	N/A	Include `mgpu = Machine(GPU);` in the generated code.
`case4`	**Execution Error:** Assertion failed: stride does not match expected value.	Memory layout is unexpected.	Adjust the layout constraints or move tasks to different processor types.
`case5`	**Execution Error:** DGEMM parameter number 8 had an illegal value	Memory layout is unexpected.	Adjust the layout constraint.
`case6`	**Execution Error:** Slice processor index out of bound	IndexTaskMap statements cause error.	Ensure that the first index of mgpu ends with `% mgpu.size[0]`, and the second element ends with `% mgpu.size[1]`.
`case7`	**Execution Error:** Assertion 'event.exists()' failed	InstanceLimit statements cause error.	Avoid generating InstanceLimit statements.
`case8`	**Performance Metric:** Execution time is 0.03s.	N/A	Move more tasks to GPU to reduce execution time.
`case9`	**Performance Metric:** Achieved throughput = 4877 GFLOPS	N/A	Try using different `IndexTaskMap` or `SingleTaskMap` statements to maximize throughput.

Table A2: Raw execution output and AutoGuide (error explanations and adjustment suggestions) for different cases.

A.5 Trace Agent Code

Trace [14] uses Python decorators like `@bundle` to annotate Python programs. It allows us to design an LLM code generation agent as if we were writing a Python program ourselves. We first set up an end-to-end runnable Python program that can generate a valid mapper program by randomly making decisions over the search space. We show the high-level structure of our Trace Mapper in Figure A3. Figure A2 shows how we incorporate the feedback from the execution to update the agent. At each optimization step, Trace will execute `DSLMapperGenerator` and collect the corresponding execution flow to build up a graph. Then it will make a call to an LLM to perform an update to any function that is decorated with `@bundle(trainable=True)`. The `DSLMapperGenerator` is structured in the same way as providing a search space specified by the DSL, where an LLM optimizer can make decisions along the pre-designed axes. We note that this type of design is only enabled by recent developments like Trace and is much more challenging to do using older LLM-based frameworks.

```
policy = MapperAgent()
params = policy.parameters()
optimizer = trace.Optimizer(params)

app  = GetApplicationInfo()
test = GetMapperEvaluator(app)

for i in range(iterations):
  # Forward pass
  try:
    mapper = policy(app)
    # feedback (str) contains performance
    feedback = test(mapper)
  except TraceExecutionError as e:
    feedback = str(e)
    target = e.exception_node

  # Backward pass and update
  optimizer.zero_feedback()
  optimizer.backward(target, feedback)
  optimizer.step()
```

Figure A2: We show how we use Trace to incorporate the feedback from the execution to update the agent, with a Pytorch-like syntax.

```python
import opto.trace as trace

class MapperAgent(trace.Module):
    @trace.bundle(trainable=True)
    def task_decision(self, tasks):
        ...

    @trace.bundle(trainable=True)
    def region_decision(self, regions):
        ...

    @trace.bundle(trainable=True)
    def layout_decision(self):
        ...

    @trace.bundle(trainable=True)
    def instance_limit_decision(self, tasks):
        ...

    @trace.bundle(trainable=True)
    def index_task_map_decision(self, index_tasks):

    @trace.bundle(trainable=True)
    def single_task_map_decision(self, single_tasks):
        ...

    def generate_mapper(self):
        """
        Generate the final mapper code by combining all code statements.
        """
        task_statements = self.task_decision(self.tasks)
        region_statements = self.region_decision(self.regions)
        layout_statements = self.layout_decision()
        instance_limit_statements = self.instance_limit_decision(self.tasks)
        index_task_map_statements =
    self.index_task_map_decision(self.index_tasks, self.index_task_specification)
        single_task_statements = self.single_task_map_decision(self.single_tasks)

        code_statements = (
            task_statements +
            region_statements +
            layout_statements +
            instance_limit_statements +
            index_task_map_statements +
            single_task_statements
        )
        # Combine all code statements and function definitions into a single
    string
        code_list = code_statements
        mapper_code = str_join(node('\n'), *code_list)
        return mapper_code
```

Figure A3: High-level structure of the Trace-based agent template, where functions annotated with `@bundle(trainable=True)` define the search space that the LLM optimizer updates during mapper generation. **Note**: This agent serves as a shared starting point for **ALL** tasks. For each task, we produce a mapper from this starting agent and then ask LLMs to "optimize" this agent (by changing functions that are `trainable`) to produce mappers that are optimal for the particular task.

A.6 Mapping Strategies

Strategy 1: Map the tasks of `calculate_new_currents`, `distribute_charge`, `update_voltages` onto GPUs in this way: linearize the 2D GPU processor space into 1D, then perform 1D block mapping from launch domain to the linearized 1D processor space.

```
Task * GPU,CPU; # for any task, run on GPU if supported
Region * *GPU FBMEM; # for any task, any region, if mapped onto GPU, use FBMEM as default
Region * * CPU SYSMEM; # if mapped onto CPU, use SYSMEM as default

Layout * * * SOA C_order;

mcpu = Machine(CPU);
mgpu = Machine(GPU);

========== Above is fixed ==========
def linearblock(Task task) {
    return mgpu[task.ipoint[0] / mgpu.size[1], task.ipoint[0] % mgpu.size[1]];
}

IndexTaskMap calculate_new_currents,distribute_charge,update_voltages linearblock;
```

Strategy 2: Place ghost/shared regions (rp_shared and rp_ghost) onto GPU zero-copy memory

```
Task * GPU,CPU; # for any task, run on GPU if supported

Region * * GPU FBMEM; # for any task, any region, if mapped onto GPU, use FBMEM as default
Region * * CPU SYSMEM; # if mapped onto CPU, use SYSMEM as default

Layout * * * SOA C_order;

mcpu = Machine(CPU);
mgpu = Machine(GPU);

========== Above is fixed ==========

Region * rp_shared GPU ZCMEM;
Region * rp_ghost GPU ZCMEM;
```

Strategy 3: Use Array Of Struct (AOS) data layout for all data instead of the default SOA

```
Task * GPU,CPU; # for any task, run on GPU if supported

Region * * GPU FBMEM; # for any task, any region, if mapped onto GPU, use FBMEM as default
Region * * CPU SYSMEM; # if mapped onto CPU, use SYSMEM as default

mcpu = Machine(CPU);
mgpu = Machine(GPU);

========== Above is fixed ==========

Layout * * * AOS;
```

Strategy 4: Use Fortran ordering of data layout for all data instead of the default C order

```
Task * GPU,CPU; # for any task, run on GPU if supported

Region * * GPU FBMEM; # for any task, any region, if mapped onto GPU, use FBMEM as default
Region * * CPU SYSMEM; # if mapped onto CPU, use SYSMEM as default

mcpu = Machine(CPU);
mgpu = Machine(GPU);

========== Above is fixed ==========

Layout * * * F_order;
```

Strategy 5: Align all the regions to 64 bytes while using the Fortran ordering of data

```
Task * GPU,CPU; # for any task, run on GPU if supported

Region * * GPU FBMEM; # for any task, any region, if mapped onto GPU, use FBMEM as default
Region * * CPU SYSMEM; # if mapped onto CPU, use SYSMEM as default

mcpu = Machine(CPU);
mgpu = Machine(GPU);

========== Above is fixed ==========

Layout * * * Align==64 F_order;
```

Strategy 6: Place the task calculate_new_currents onto CPU

```
Task * GPU,CPU; # for any task, run on GPU if supported

Region * * GPU FBMEM; # for any task, any region, if mapped onto GPU, use FBMEM as default
Region * * CPU SYSMEM; # if mapped onto CPU, use SYSMEM as default

mcpu = Machine(CPU);

mgpu = Machine(GPU);

Layout * * * SOA C_order;

========== Above is fixed ==========
Task calculate_new_currents CPU;
```

Strategy 7: Collect all the memory used by task calculate_new_currents

```
Task * GPU,CPU; # for any task, run on GPU if supported

Region * * GPU FBMEM; # for any task, any region, if mapped onto GPU, use FBMEM as default
Region * * CPU SYSMEM; # if mapped onto CPU, use SYSMEM as default

mcpu = Machine(CPU);
mgpu = Machine(GPU);

Layout * * * SOA C_order;

========== Above is fixed ==========
CollectMemory calculate_new_currents *;
```

Strategy 8: Ensure that at most 4 tasks of calculate_new_currents can be run at the same time

```
Task * GPU,CPU; # for any task, run on GPU if supported

Region * * GPU FBMEM; # for any task, any region, if mapped onto GPU, use FBMEM as default
Region * * CPU SYSMEM; # if mapped onto CPU, use SYSMEM as default

mcpu = Machine(CPU);
mgpu = Machine(GPU);

Layout * * * SOA C_order;

========== Above is fixed ==========
InstanceLimit calculate_new_currents 4;
```

Strategy 9: Map the second region argument of task distribute_charge onto GPU's Zero-Copy memory

```
Task * GPU,CPU; # for any task, run on GPU if supported

Region * * GPU FBMEM; # for any task, any region, if mapped onto GPU, use FBMEM as default
Region * * CPU SYSMEM; # if mapped onto CPU, use SYSMEM as default

mcpu = Machine(CPU);
mgpu = Machine(GPU);

Layout * * * SOA C_order;

========== Above is fixed ==========
Region distribute_charge 1 GPU ZCMEM;
```

Strategy 10: Map the tasks of calculate_new_currents,distribute_charge,update_voltages onto GPUs in a 1D cyclic manner: perform a cyclic distribution over both the node and processor dimensions.

```
Task * GPU,CPU; # for any task, run on GPU if supported

Region * * GPU FBMEM; # for any task, any region, if mapped onto GPU, use FBMEM as default
Region * * CPU SYSMEM; # if mapped onto CPU, use SYSMEM as default

mcpu = Machine(CPU);
mgpu = Machine(GPU);

Layout * * * SOA C_order;

========== Above is fixed ==========
def cyclic1d(Task task) {
    ip = task.ipoint;
    # cyclic over node, cyclic over gpu
    return mgpu[ip[0] % mgpu.size[0], ip[0] / mgpu.size[0] % mgpu.size[1]];
}

IndexTaskMap calculate_new_currents,distribute_charge,update_voltages cyclic1d;
```

A.7 Generated Mapper Examples

Here we provide examples of generated mappers for a subset of problems. The mappers, written in DSL, are produced by the mapper agent. While the LLM is responsible for creating and refining the mapper agent, the agent itself is implemented in Python, and it generates mappers as DSL programs. For the Circuit Simulation benchmark, the optimized mapper (Figure A5) is more concise than the initial version (Figure A4), with an additional constraint for byte alignment in the data layout. In contrast, for Solomonik's algorithm, the initial mapper is relatively simple (Figure A6), whereas the final optimized mapper adopts a more complex and detailed index mapping strategy (Figure A7).

```
Task * GPU,OMP,CPU;
Task calculate_new_currents GPU;
Task update_voltages GPU;
Region * * GPU FBMEM;
Region * * * SOCKMEM,SYSMEM;
Region * all_times GPU FBMEM;
Region * all_nodes GPU FBMEM;
Region * all_wires GPU FBMEM;
Region * ghost_ranges GPU FBMEM;
Region * rp_all_nodes GPU FBMEM;
Region * all_private GPU FBMEM;
Region * all_shared GPU FBMEM;
Region * rp_shared GPU FBMEM;
Region * rp_wires GPU FBMEM;
Region * rp_ghost_ranges GPU FBMEM;
Layout * * * C_order AOS;
mgpu = Machine(GPU);

m_2d = Machine(GPU);
def same_point(Task task) {
    return m_2d[*task.parent.processor(m_2d)];
}
```

Figure A4: For the Circuit task, we show the mapper produced by the mapper agent at iteration 2.

```
Task * GPU,OMP,CPU;
Task calculate_new_currents GPU;
Task update_voltages GPU;
Region * * GPU FBMEM;
Layout * * * C_order AOS Align==128;
mgpu = Machine(GPU);

m_2d = Machine(GPU);
def same_point(Task task) {
    return m_2d[*task.parent.processor(m_2d)];
}
```

Figure A5: For the Circuit task, we show the mapper produced by the mapper agent at iteration 10.

```
Task * GPU,OMP,CPU;
Region * * GPU FBMEM;
Region * * * SOCKMEM,SYSMEM;
Layout * * * F_order SOA;
mgpu = Machine(GPU);

def block1d(Task task) {
    ip = task.ipoint;
    return mgpu[ip[0] % mgpu.size[0], ip[0] % mgpu.size[1]];
}

IndexTaskMap task_2 block1d;

m_2d = Machine(GPU);
def same_point(Task task) {
    return m_2d[*task.parent.processor(m_2d)];
}
```

Figure A6: For Solomonik's algorithm, we show the mapper produced by the mapper agent at iteration 2.

```
Task * GPU,OMP,CPU;
Region * * GPU FBMEM;
Region * * * SOCKMEM,SYSMEM;
Layout * * * C_order SOA No_Align;
mgpu = Machine(GPU);

def block1d(Task task) {
    ip = task.ipoint;
    return mgpu[ip[0] % mgpu.size[0], ip[0] % mgpu.size[1]];
}

IndexTaskMap task_1 block1d;

def cyclic1d(Task task) {
    ip = task.ipoint;
    linearize = ip[0] * 2 + ip[1];
    return mgpu[ip[0] % mgpu.size[0], linearize % mgpu.size[1]];
}

IndexTaskMap task_1 cyclic1d;

def cyclic2d(Task task) {
    ip = task.ipoint;
    linearize = ip[0] + ip[1] * 2;
    return mgpu[ip[0] % mgpu.size[0], linearize % mgpu.size[1]];
}

IndexTaskMap task_1 cyclic2d;

def linearize3D(Task task) {
    ip = task.ipoint;
    linearize = ip[0] + ip[1] + ip[2];
    return mgpu[linearize % mgpu.size[0], linearize % mgpu.size[1]];
}

IndexTaskMap task_1 linearize3D;

def linearize2D(Task task) {
    ip = task.ipoint;
    linearize = ip[0] * 2 + ip[2];
    return mgpu[linearize % mgpu.size[0], linearize % mgpu.size[1]];
}

IndexTaskMap task_1 linearize2D;
IndexTaskMap task_2 block1d;
IndexTaskMap task_2 cyclic1d;
IndexTaskMap task_2 cyclic2d;
IndexTaskMap task_2 linearize3D;
IndexTaskMap task_2 linearize2D;
IndexTaskMap task_3 block1d;
IndexTaskMap task_3 cyclic1d;
IndexTaskMap task_3 cyclic2d;
IndexTaskMap task_3 linearize3D;
IndexTaskMap task_3 linearize2D;
IndexTaskMap task_5 block1d;
IndexTaskMap task_5 cyclic1d;
IndexTaskMap task_5 cyclic2d;
IndexTaskMap task_5 linearize3D;
IndexTaskMap task_5 linearize2D;

m_2d = Machine(GPU);
def same_point(Task task) {
    return m_2d[*task.parent.processor(m_2d)];
}
```

Figure A7: For Solomonik's algorithm, we show the mapper produced by the mapper agent at iteration 10.