Astra: A Multi-Agent System for GPU Kernel Performance Optimization

Anonymous Author(s)
Affiliation
Address
email

Abstract

GPU kernel optimization has long been a central challenge at the intersection of high-performance computing and machine learning. Efficient kernels are crucial for accelerating large language model (LLM) training and serving, yet attaining high performance typically requires extensive manual tuning. Compiler-based systems reduce some of this burden, but still demand substantial manual design and engineering effort. Recently, researchers have explored using LLMs for GPU kernel generation, though prior work has largely focused on translating high-level PyTorch modules into CUDA code. In this work, we introduce Astra, the first LLM-based multi-agent system for GPU kernel optimization. Unlike previous approaches, Astra starts from existing CUDA implementations extracted from SGLang, a widely deployed framework for serving LLMs, rather than treating PyTorch modules as the specification. Within Astra, specialized LLM agents collaborate through iterative code generation, testing, profiling, and planning to produce kernels that are both correct and high-performance. On kernels from SGLang, Astra achieves an average speedup of 1.32× using zero-shot prompting with OpenAI o4-mini. A detailed case study further demonstrates that LLMs can autonomously apply loop transformations, optimize memory access patterns, exploit CUDA intrinsics, and leverage fast math operations to yield substantial performance gains. Our work highlights multi-agent LLM systems as a promising new paradigm for GPU kernel optimization.

1 Introduction

Recent advances in large language models (LLMs) have led to state-of-the-art performance on a wide range of tasks, including reasoning and code generation [1–6]. Building on these capabilities, autonomous agents powered by LLMs have begun to automate parts of the software development pipeline [7–9]. In this work, we investigate the application of LLM-powered agents to GPU kernel optimization, a long-standing challenge at the intersection of high-performance computing and machine learning that requires generating code that is both correct and highly optimized.

GPU kernel optimization is essential for improving the efficiency of LLM serving and training, which is critical for the successful deployment of LLMs. However, even with decades of advances in GPU programming, kernel development remains a fundamentally difficult real-world engineering problem. Rapid hardware evolution often requires extensive manual tuning and reimplementation. For example, FlashAttention-2 [10] suffered a 47% performance drop when first ported to NVIDIA's H100 GPUs, and it was only after more than two years that FlashAttention-3 [11] introduced substantial new optimizations to recover performance. In addition, emerging model architectures [12–15] and dynamic workloads with variable input lengths [16] further complicate kernel optimization. As a

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

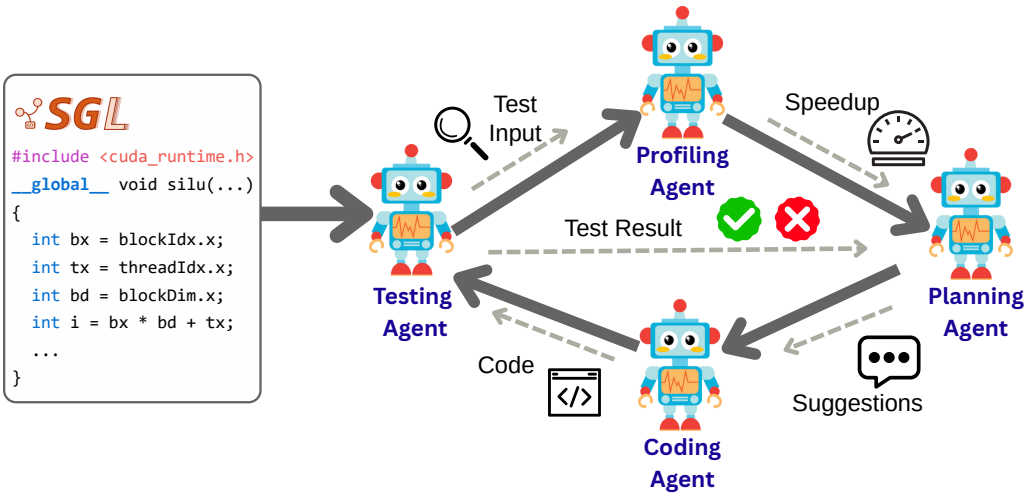

Figure 1: Overview of Astra. Given an existing GPU kernel extracted from SGLang, Astra employs a multi-agent approach for kernel optimization, where specialized LLM agents collaborate through iterative code generation, testing, profiling, and planning to produce correct and high-performance kernels.

result, many available kernel implementations operate well below hardware peak. Closing this gap is vital for advancing performance, reducing costs, and improving energy efficiency.

From a systems perspective, there are two dominant paradigms for GPU kernel optimization. The first is fully manual tuning, exemplified by libraries such as NVIDIA cuDNN [17]. This approach demands extensive manual effort, involves time-consuming engineering cycles, and can still leave optimization opportunities untapped. The second paradigm is compiler-based optimization, represented by systems such as TVM [18], Triton [19], Mirage [20], ThunderKittens [21], and others [22, 23]. For instance, Triton introduces a tile-level intermediate representation combined with autotuning to deliver performance close to hand-optimized kernels, significantly reducing the engineering burden for end users. Nevertheless, these compiler-based systems themselves require substantial engineering effort to develop and must be continuously adapted as hardware evolves.

Given the significant potential of LLMs, researchers have actively explored their use for GPU kernel optimization. KernelBench [24] is a pioneering work that first formulates the task for LLMs and introduces a corresponding benchmark. Other studies have explored single-agent approaches [25–27] as well as training-based methods for improving LLMs [28, 29].

In this work, we introduce Astra, the first LLM-based multi-agent system for GPU kernel optimization. Our key observation is that kernel optimization is inherently a multi-stage process that includes code generation, testing, profiling, and planning, and a single LLM agent is unlikely to excel at all of these tasks. As shown in Figure 1, Astra addresses this challenge by decomposing the task into specialized agents that collaborate iteratively. This coordinated workflow leverages complementary agent capabilities, enabling systematic exploration of the optimization space and consistently producing kernels that are both correct and high-performance.

Our setting contrasts with KernelBench [30] in two important respects. Unlike KernelBench, which frames the task as generating CUDA kernels from high-level PyTorch models written in Python, we focus on optimizing existing CUDA implementations. This reflects the reality of production environments, where kernels are already available and the real challenge is squeezing out additional performance rather than generating CUDA code from scratch. KernelBench has already demonstrated that translation from Python to CUDA is non-trivial for LLMs; our work focuses squarely on performance optimization and avoids the additional burden of translation, which can introduce errors and degrade performance. In addition, our kernels are taken directly from SGLang [31] and can be seamlessly reintegrated into the system. As SGLang is a production-grade LLM serving framework deployed at scale and responsible for generating trillions of tokens per day across major enterprises and institutions, even modest kernel-level improvements can yield substantial real-world impact.

We evaluate Astra on three kernels extracted from SGLang [31] and observe an average speedup of 1.32× using zero-shot prompting with OpenAI o4-mini. Importantly, these results are achieved without any additional training, including supervised fine-tuning or reinforcement learning, which highlights the effectiveness of our approach in a pure prompting setting and suggests further potential when combined with training-based methods. To demonstrate the necessity of dedicated agent roles, we compare against a single-agent baseline, which attains only 1.08× speedup on average. Finally, we conduct an in-depth case analysis to investigate the source of performance gains. Our findings show that LLMs can autonomously apply loop transformations, restructure memory access patterns, make extensive use of CUDA intrinsics, and exploit fast math operations, all of which contribute to the observed speedups.

In summary, our contributions are:

- We design and implement Astra, a multi-agent system for GPU kernel optimization, in which specialized LLM agents collaborate through iterative code generation, testing, profiling, and planning to produce correct and high-performance kernels.
- We demonstrate an average speedup of 1.32× on kernels from SGLang, a production-grade LLM serving framework, and our optimized kernels can be seamlessly reintegrated to deliver substantial real-world impact.
- We conduct a detailed manual analysis of the kernels generated by Astra and identify the optimization strategies, including loop transformations, memory access improvements, extensive use of CUDA intrinsics, and faster math operations, that account for the observed speedups.

2 Related Work

Multi-Agent Systems Multi-agent systems (MAS) consist of multiple interacting agents that collaborate to solve complex, shared problems that exceed the capabilities of a single agent. This paradigm is particularly well-suited to programming, where intricate workflows can be naturally decomposed into sub-tasks such as planning, implementation, testing, and profiling. Recent work has explored multi-agent frameworks including AutoGen [32], Trace [33], and MetaGPT [34, 35], which have demonstrated strong performance on benchmarks in mathematics and code generation [36–39, 9]. However, there has been little exploration of applying MAS to GPU kernel optimization, a domain where highly specialized performance considerations introduce unique challenges.

Compiler and Learning-Based Approaches to GPU Kernel Optimization GPU kernel optimization has long been driven by compiler frameworks and domain-specific languages (DSLs). Systems such as Halide [40], TVM [18], MLIR [41], TensorFlow XLA [42], and NVIDIA CUTLASS [43], along with others [44–48], provide high-level abstractions for expressing tensor computations and support compiler-driven optimizations. To further improve performance, autotuning frameworks such as AutoTVM [18], Ansor [22], and AMOS [23] leverage search and machine learning to explore large optimization spaces. More recent systems, including Triton [19], Mirage [20], and ThunderKittens [21], expand on these ideas. For example, Triton introduces a tile-level intermediate representation and autotuning, achieving performance close to hand-optimized kernels. Nevertheless, compiler-based approaches often fall short of expert-level performance without extensive tuning, and generalization across hardware platforms remains difficult [21]. Despite their progress, these systems are still constrained by rigid compilation pipelines and require significant engineering effort to build.

LLM-Driven Approaches to High-Performance Code Generation Early efforts in LLM-based code generation, such as AlphaCode [49], primarily targeted general-purpose programming tasks and demonstrated promising results. More recently, research has increasingly focused on domain-specific high-performance code generation, spanning tasks such as vectorization [50, 51], assembly-level optimization [52], parallel programming with domain-specific languages (DSLs) [53, 39, 54], and tensor program optimization [55]. A particularly active direction is the automatic generation of performant GPU kernels [24, 56]. Because code optimization provides verifiable rewards, iterative refinement has emerged as a natural paradigm: models generate candidate kernels and progressively improve them through feedback loops involving compilation checks, correctness validation, runtime profiling, or self-reflection [24, 57, 39]. Unlike general code generation tasks, a central challenge in

code optimization is to ensure that LLMs generate code that is both functionally correct and highly optimized, where correctness means equivalence to the original program for all inputs [58, 59]. To tackle this challenge, researchers have explored both prompt-based approaches [60, 61, 26, 27, 25, 62] and training-based methods, including multi-turn reinforcement learning [28] and contrastive reinforcement learning [29]. Our work addresses this challenge by adopting a multi-agent system approach.

3 Method

3.1 Task Definition

The goal of CUDA optimization is to produce an optimized kernel S' that runs faster than the baseline kernel S while preserving its functional correctness. Below, we formally define the correctness and performance criteria, and then outline how our setup differs from prior work.

Correctness. Let $\mathcal{X}$ be the input domain and $\mathcal{Y}$ the output space. The baseline and optimized kernels are functions $S, S' : \mathcal{X} \to \mathcal{Y}$. Ideally, we require

$$\forall x \in \mathcal{X} : \; S'(x) = S(x),$$

or, allowing floating-point deviations,

$$\forall x \in \mathcal{X} : \; d\big(S'(x), S(x)\big) \leq \varepsilon,$$

for a discrepancy metric d and tolerance $\varepsilon \geq 0$. Since exact equivalence is undecidable in practice, we evaluate correctness on a finite test suite

$$T = \{(x_i, y_i)\}_{i=1}^{m}, \qquad y_i := S(x_i),$$

where the x_i are chosen to represent diverse tensor shapes and values. We deem S' correct if

$$\max_{1 \leq i \leq m} d\big(S'(x_i), y_i\big) \leq \varepsilon.$$

Performance. Let $\tau(S, x)$ denote the runtime of kernel S on input $x \in \mathcal{X}$. For each input x, the speedup is

$$\sigma(x) = \frac{\tau(S, x)}{\tau(S', x)}.$$

To summarize results over the test suite T, we report the geometric mean σ_T, which is the standard choice for averaging speedups because it correctly aggregates ratios, is symmetric between speedups and slowdowns, and reduces the influence of outliers:

$$\sigma_T = \left(\prod_{i=1}^{m} \frac{\tau(S, x_i)}{\tau(S', x_i)} \right)^{1/m}.$$

The optimization objective is to maximize this geometric-mean speedup while preserving correctness.

3.2 Multi-Agent System

Agent Roles. As shown in Figure 1, Astra is organized around four specialized agents, each responsible for a distinct stage of the CUDA optimization pipeline. The *testing agent* creates a suite of test cases from the baseline kernel and checks the correctness of candidate kernels. The *profiling agent* measures execution time on the test suite, providing performance feedback. The *planning agent* combines correctness and performance signals to propose targeted modifications. The *coding agent* applies these suggestions to generate new kernel implementations. Together, these agents form a feedback loop that supports iterative refinement while preserving correctness.

Algorithm. Algorithm 1 outlines the multi-agent optimization procedure. The process begins with the construction of an initial test suite and profiling of the baseline kernel. The system then proceeds through R iterative rounds: in each round, the planning agent proposes modifications, the coding agent generates a new candidate kernel, and the testing and profiling agents re-evaluate correctness and performance. All results are recorded in a log of tuples $(\textit{round}, \textit{code}, \textit{correctness}, \textit{performance})$, where *correctness* is a binary indicator of whether the candidate passes the tests. This log enables systematic tracking of the optimization trajectory.

Algorithm 1 Multi-Agent CUDA Optimization

Input: Baseline CUDA code S_0, number of rounds R
Define: `TestingAgent` ▷ Generate or run tests.
`ProfilingAgent` ▷ Measure performance.
`PlanningAgent` ▷ Propose suggestions given correctness and performance signals.
`CodingAgent` ▷ Apply suggestions to previous code S_{prev}
`Log` ▷ List of $(\textit{round}, \textit{code}, \textit{correctness}, \textit{performance})$ for all iterations.
`Test suite` T ▷ Tests generated by `TestingAgent`.
Output: `Log`

$T \leftarrow$ `TestingAgent.GenerateTests`(S_0) ▷ Initialization
$\mathsf{perf}_0 \leftarrow$ `ProfilingAgent.Profile`(S_0, T)
$\mathsf{Log} \leftarrow [\,]$
`Append`$(\mathsf{Log}, (0, S_0, \mathsf{True}, \mathsf{perf}_0))$
$S_{\text{prev}} \leftarrow S_0$
$\mathsf{pass}_{\text{prev}} \leftarrow \mathsf{True}$
$\mathsf{perf}_{\text{prev}} \leftarrow \mathsf{perf}_0$
for $r \leftarrow 1$ to R **do** ▷ Iterative optimization starts
  $\mathsf{suggestions} \leftarrow$ `PlanningAgent.Suggest`$(S_{\text{prev}}, \mathsf{pass}_{\text{prev}}, \mathsf{perf}_{\text{prev}})$
  $S_{\text{new}} \leftarrow$ `CodingAgent.Apply`$(S_{\text{prev}}, \mathsf{suggestions})$
  $\mathsf{pass}_{\text{new}} \leftarrow$ `TestingAgent.Validate`(S_{new}, T)
  $\mathsf{perf}_{\text{new}} \leftarrow$ `ProfilingAgent.Profile`(S_{new}, T)
  `Append`$(\mathsf{Log}, \ (r, S_{\text{new}}, \mathsf{pass}_{\text{new}}, \mathsf{perf}_{\text{new}}))$
  $S_{\text{prev}} \leftarrow S_{\text{new}}$
  $\mathsf{pass}_{\text{prev}} \leftarrow \mathsf{pass}_{\text{new}}$
  $\mathsf{perf}_{\text{prev}} \leftarrow \mathsf{perf}_{\text{new}}$
return *Log*

Pre-Processing and Post-Processing. Allowing Astra to directly optimize the raw CUDA kernels in the SGLang framework [31] is difficult because these kernels have many internal dependencies. To address this, we perform a manual pre-processing step: extracting and simplifying the kernels into stand-alone versions that serve as the baseline inputs for Astra. After optimization, we apply a post-processing step that integrates the generated kernels back into SGLang and validates them against the original framework implementation (rather than only the extracted version). We report speedups relative to the original SGLang kernels, ensuring that the optimized kernels can be seamlessly integrated into the framework as drop-in replacements and that performance is measured within the full framework.

4 Experimental Setup

Metrics. Our evaluation focuses on both correctness and performance. Correctness is determined using test cases that we construct with diverse tensor input shapes. We compare the outputs of generated kernels against the execution results of the original SGlang implementation, which serves as the ground truth. For performance, we measure the execution time of both the original kernel and the optimized version on the same tensor shapes, and report speedup as the metric. While the multi-agent framework internally produces its own test cases through the testing agent, the final evaluation relies on manually designed test cases to ensure high confidence in functional validation.

Kernels. We evaluate three kernels from the LLM serving framework SGLang [31]: `silu_and_mul`, `fused_add_rmsnorm`, and `merge_attn_states_lse`. Their computations are summarized in Table 1.

Performance Measurement. We evaluate performance across a range of input shapes and report average results. For each input shape, we run 100 repetitions after 20 warm-up runs. The input shapes are selected based on the actual dimensions used in modern LLMs, including the LLaMA-7B, 13B, and 70B models. A detailed analysis of how input shapes affect performance is provided in Section 6.1.

Index	Kernel Name	Computation
Kernel 1	`merge_attn_states_lse`	$\mathbf{V}_{\text{out}} = \frac{e^{S_a}\,\mathbf{V}_a + e^{S_b}\,\mathbf{V}_b}{e^{S_a} + e^{S_b}}$, $S_{\text{out}} = \log\left(e^{S_a} + e^{S_b}\right)$
Kernel 2	`fused_add_rmsnorm`	$\mathbf{y} = \frac{\mathbf{x} + \mathbf{r}}{\sqrt{\frac{1}{D}\lVert\mathbf{x} + \mathbf{r}\rVert_2^2 + \varepsilon}} \odot \mathbf{w}$
Kernel 3	`silu_and_mul`	$\mathbf{out} = \text{SiLU}(\mathbf{x}) \odot \mathbf{g}$, $\text{SiLU}(z) = \frac{z}{1+e^{-z}}$

Table 1: Kernel names and computations.

Kernel	LoC-Base	LoC-Opt.	ΔLoC	Time-Base	Time-Opt.	**Speedup**	**Correct**
Kernel 1	124	232	+87%	31.4	24.9	1.26×	✓
Kernel 2	108	163	+50%	41.3	33.1	1.25×	✓
Kernel 3	99	157	+59%	20.1	13.8	1.46×	✓
Average	110	184	+64%	30.9	23.9	1.32×	✓

Table 2: Baseline vs. optimized kernels: Lines of Code (LoC) and execution time (μs). All kernels optimized by our multi-agent system are correct.

Implementation. We implement our multi-agent system with the OpenAI Agents SDK framework [63], which offers standardized abstractions for defining agents and integrating function tools. The agents are powered by OpenAI's `o4-mini` model, and all experiments are conducted on a machine equipped with NVIDIA H100 GPUs. We set the number of rounds to optimize R to be 5.

5 Results

5.1 Main Results

Correctness. As shown in the last column of Table 2, all three optimized kernels are validated against the original SGLang implementations and confirmed to be correct. As described in Section 4, we do not rely on test cases generated by the testing agent for functional validation. Instead, we manually construct test cases for the kernels produced by Astra and check their outputs against the original SGLang kernels.

Performance. Table 2 summarizes the performance gains achieved by Astra across the three kernels. The results show that Astra can consistently improve performance while preserving correctness. For `merge_attn_states_lse` (Kernel 1), the optimized version has 87% more lines of code and delivers a 1.26× speedup. For `fused_add_rmsnorm` (Kernel 2), the optimized kernel contains 50% more lines and achieves a 1.25× speedup. For `silu_and_mul` (Kernel 3), the optimized kernel has 59% more lines and yields a 1.46× speedup. Overall, with only five optimization rounds, Astra achieves an average speedup of 1.32× and up to 1.46×, measured over a set of representative tensor shapes. We present detailed case studies in Section 5.3 and analyze how tensor shapes influence performance in Section 6.1.

5.2 Comparison with Single-Agent Approach

Setup of Single-Agent Method. In the single-agent setting, we continue to use the OpenAI Agents SDK framework but instantiate only one agent. This agent handles all tasks, including testing, profiling, planning, and code generation, and has access to the same set of tools as in the multi-agent setting. For fairness, we run the same number of optimization rounds, set to five, and the only difference lies in the number of agents involved.

Kernel	Time-Base (μs)	Correct - SA	Speedup - SA	Correct - MA	Speedup - MA
Kernel 1	31.4	✓	0.73×	✓	1.26×
Kernel 2	41.3	✓	1.18×	✓	1.25×
Kernel 3	20.1	✓	1.48×	✓	1.46×
Average	30.9	✓	1.08×	✓	1.32×

Table 3: Single-Agent (SA) vs. Multi-Agent (MA) comparison: baseline runtime (Time-Base), correctness, and speedup (×).

```
// two scalar scores
float sa = score_a, sb = score_b;
// inner loop
for (int d = 0; d < D; ++d)
{
  float smax = fmaxf(sa, sb); // repeated
  float wa   = expf(sa - smax); // repeated
  float wb   = expf(sb - smax); // repeated
  float inv  = 1.0f / (wa + wb + 1e-12f);
  float a = wa * inv, b = wb * inv;
  out[d] = a * va[d] + b * vb[d];
}
```

(a) Baseline: recompute inside the inner loop

```
// compute once per output vector
float sa = score_a, sb = score_b;
float smax = fmaxf(sa, sb);
float wa = expf(sa - smax), wb = expf(sb - smax);
float inv = 1.0f / (wa + wb + 1e-12f);
float a = wa * inv;
float b = wb * inv;

// lightweight inner loop
for (int d = 0; d < D; ++d) {
  out[d] = a * va[d] + b * vb[d];
}
```

(b) Optimized: hoist loop-invariant computations

Figure 2: Hoisting loop-invariant computation in `merge_attn_states_lse`.

Performance. As shown in Table 3, the multi-agent approach achieves higher performance speedup than the single-agent approach (1.32× vs. 1.08×), while both approaches consistently generate correct kernels. We observe that the advantages of the multi-agent setup become more pronounced as kernel complexity increases. For kernel 3, which is relatively simple, the performance of both approaches is comparable. In contrast, kernel 1 is the most complex and exposes the limitations of the single-agent setup, where certain tasks may not be carried out effectively enough to yield good overall results. In particular, the slowdown of Kernel 1 under the single-agent setting was due to unrepresentative test inputs generated during test construction, which biased the profiling results. This issue does not occur in the multi-agent approach, where one agent is dedicated to generating representative test inputs and another to conducting profiling. Overall, these findings demonstrate that Astra provides greater advantages over the single-agent setup when handling more complex kernels.

5.3 Case Studies

We compare the source code of the baseline and Astra-optimized kernels and conduct detailed performance profiling with NVIDIA Nsight Compute. Overall, the speedups stem from eliminating redundant computation, improving memory-access efficiency, and exploiting advanced CUDA features. Concretely, the optimized kernels apply loop transformations to enhance parallelism, adopt more aggressive memory-access strategies to maximize bandwidth, make extensive use of CUDA intrinsics for hardware-level efficiency, and leverage fast math operations. The following examples illustrate these optimization strategies.

Kernel 1: `merge_attn_states_lse` A key optimization shown in Figure 2 is hoisting loop-invariant computations out of the inner element loop. In the baseline, the mixing weights and their normalization are recomputed for every element of the output vector, incurring repeated exponentials and a division within the hot loop. The optimized version computes these quantities once per output vector, leaving the inner loop with only memory loads, multiply–add, and a store. By removing expensive operations from the loop body, the optimized kernel lowers the instruction count and increases throughput without affecting correctness.

Kernel 2: `fused_add_rmsnorm` This kernel contains a block-level reduction that dominates runtime. As shown in Figure 3, the baseline implements a tree-based reduction in on-chip shared memory, which already improves latency and bandwidth relative to a naive global-memory reduction,

```cpp
/* tx = threadIdx.x, BS = BLOCK_SIZE */
__shared__ float sm[BS];

float s = ...;        // per-thread sum
sm[tx] = s;
__syncthreads();

for (int off = BS/2; off > 0; off >>= 1) {
  if (tx < off)
    sm[tx] += sm[tx + off];
  __syncthreads();
}
...
```

(a) Baseline: shared-memory tree reduction

```cpp
/* lane = tx & 31, warp = tx >> 5 */
float s = ...;        // per-thread sum

unsigned m = 0xffffffffu;      // intra-warp
for (int off = 16; off > 0; off >>= 1)
  s += __shfl_down_sync(m, s, off);

__shared__ float ws[BS/32];    // one per warp
if (lane == 0)
  ws[warp] = s;
__syncthreads();

...
```

(b) Optimized: warp-level shuffle, brief shared-memory finalize

Figure 3: Reduction strategies in `fused_add_rmsnorm`. Figure 3a: block-level tree reduction in shared memory with synchronization each step. Figure 3b: intra-warp reduction in registers using `__shfl_down_sync`, followed by a short inter-warp aggregation in shared memory.

```cpp
...
const __half* x_ptr = row_in;
__half xv = x_ptr[vec_idx];
...
```

(a) Baseline: scalar half-precision

```cpp
...
__half2* x2 = reinterpret_cast<__half2*>(row_in);
__half2 xv2 = x2[vec_idx];
...
```

(b) Optimized: half2 vectorized load

Figure 4: Comparison of global-memory loads in the baseline and optimized kernels. The baseline uses a scalar half-precision load, while the optimized version employs a vectorized half2 load for improved efficiency.

```cpp
__device__ float silu_f(float x)
{
    return x / (1.0f + expf(-x));
}
```

(a) Baseline: standard library math + division

```cpp
__device__ float silu_fastf(float x) {
    float y = __expf(-x);
    float r = __frcp_rn(1.0f + y);
    return __fmul_rn(x, r);
}
```

(b) Optimized: fast-math intrinsics

Figure 5: Side-by-side SiLU implementations. The optimized kernel replaces a division with a reciprocal–multiply sequence and uses the fast exponential intrinsic, improving compute throughput.

but progressively disables threads as the reduction proceeds. The optimized version first performs an intra-warp reduction using warp-level intrinsics (`__shfl_down_sync`), which keeps partial sums in registers and reduces synchronization overhead. The remaining inter-warp reduction is then completed in shared memory. This register-resident intra-warp phase, followed by a short shared-memory phase, yields higher arithmetic throughput and lower memory traffic than the shared-memory-only approach.

Kernel 3: `silu_and_mul` We highlight two key optimization strategies: vectorized memory access and the use of fast math intrinsics. As shown in Figure 4, the baseline kernel performs scalar loads, fetching each `__half` value individually from global memory. In contrast, the optimized kernel employs vectorized loads by grouping two contiguous FP16 values into a `__half2` type, allowing each instruction to retrieve a pair of elements simultaneously. This reduces the number of memory transactions and increases effective memory bandwidth. Similar vectorized access patterns are also applied in Kernel 1 and Kernel 2.

Beyond memory access, compute throughput is further improved through an optimized SiLU implementation. The baseline computes SiLU using standard math library calls and a floating-point division (Figure 5a). The optimized kernel (Figure 5b) instead uses CUDA device intrinsics: `__expf` for exponentiation, `__frcp_rn` for reciprocal, and `__fmul_rn` for multiplication. Replacing the division with a reciprocal–multiplication sequence reduces instruction latency, improves arithmetic pipeline utilization, and achieves faster execution while preserving numerical correctness.

Kernel	Shapes	Time-Base (μs)	Time-Opt. (μs)	Speedup
Kernel 1	[512, 32, 256]	32.9	22.6	1.46x
	[512, 40, 128]	32.4	20.6	1.57x
	[768, 32, 256]	32.5	32.5	1.00x
	[512, 64, 128]	32.0	28.2	1.14x
Kernel 2	[256, 4096]	24.3	18.3	1.33x
	[1024, 4096]	34.0	28.3	1.20x
	[128, 11008]	25.0	19.4	1.28x
	[512, 14336]	46.1	43.0	1.07x
Kernel 3	[16, 4096]	20.9	14.2	1.47
	[32, 5120]	20.3	13.7	1.49
	[64, 8192]	20.3	13.5	1.50
	[16, 12288]	20.4	13.6	1.50

Table 4: Impact of tensor shapes on performance.

6 Discussion

6.1 Impact of Tensor Shapes on Performance Speedup

To study the effect of tensor shapes on performance, we report results for four representative shapes for each kernel. As shown in Table 4, the kernels optimized by Astra achieve consistent speedups across different shapes. For `merge_attn_states_lse` (kernel 1), we use shapes of the form [seq_len, number_of_heads, head_dim]; for `fused_add_rmsnorm` (kernel 2) and `silu_and_mul` (kernel 3), we use [batch_size, hidden_size]. Since performance speedup varies with tensor shape, in Section 5 we report the average speedup for each kernel across a set of common shapes drawn from widely used open-source models, ensuring that the results generalize across diverse shapes and serving scenarios.

Unlike tensor compiler optimization approaches [22, 23], which perform shape-specific tuning, Astra does not prompt agents to optimize for a particular shape. Instead, it aims to deliver performance improvements for general tensor computations.

6.2 Limitations and Future Work

Our evaluation currently focuses on three CUDA kernels, and the framework is tailored to SGLang [31]. In future work, we aim to extend support to a broader set of kernels and additional frameworks such as vLLM [64], PyTorch [65], and TorchTitan [66].

A key limitation is that the pre-processing and post-processing steps (Section 3.2) are fully manual. Pre-processing requires extracting and simplifying kernels into stand-alone versions suitable as inputs to Astra, while post-processing involves monkey-patching the optimized kernels back into SGLang and validating them against the original implementation. These steps are non-trivial to automate due to the complexity of modern serving frameworks. Future research should explore how to make this process more automated, potentially with human-in-the-loop guidance, so that Astra can scale to larger sets of kernels.

7 Conclusion

GPU kernel optimization is a critical yet labor-intensive challenge in high-performance computing and machine learning. In this work, we introduced Astra, the first LLM-based multi-agent system designed specifically for GPU kernel optimization. Unlike prior approaches that translate high-level PyTorch modules into CUDA code, Astra operates directly on existing CUDA kernels from SGLang, a widely deployed LLM serving framework. By coordinating specialized agents for code generation, testing, profiling, and planning, Astra produces kernels that are both correct and high-performance. Our evaluation shows that Astra delivers an average speedup of 1.32×, with case studies highlighting how LLMs can autonomously apply loop transformations, restructure memory access, exploit CUDA intrinsics, and leverage fast math operations. These results underscore the promise of multi-agent LLM systems as a new paradigm for kernel performance optimization.

References

[1] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[2] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[3] A. Wei, H. Tan, T. Suresh, D. Mendoza, T. S. Teixeira, K. Wang, C. Trippel, and A. Aiken, "Vericoder: Enhancing llm-based rtl code generation through functional correctness validation," *arXiv preprint arXiv:2504.15659*, 2025.

[4] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song *et al.*, "Measuring coding challenge competence with apps," *arXiv preprint arXiv:2105.09938*, 2021.

[5] J. Ahn, R. Verma, R. Lou, D. Liu, R. Zhang, and W. Yin, "Large language models for mathematical reasoning: Progresses and challenges," *arXiv preprint arXiv:2402.00157*, 2024.

[6] A. Wei, Y. Wu, Y. Wan, T. Suresh, H. Tan, Z. Zhou, S. Koyejo, K. Wang, and A. Aiken, "Satbench: Benchmarking llms' logical reasoning via automated puzzle generation from sat formulas," *arXiv preprint arXiv:2505.14615*, 2025.

[7] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?" *arXiv preprint arXiv:2310.06770*, 2023.

[8] N. Jain, M. Shetty, T. Zhang, K. Han, K. Sen, and I. Stoica, "R2e: Turning any github repository into a programming agent environment," in *Forty-first International Conference on Machine Learning*, 2024.

[9] J. Yang, C. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "Swe-agent: Agent-computer interfaces enable automated software engineering," *Advances in Neural Information Processing Systems*, vol. 37, pp. 50 528–50 652, 2024.

[10] T. Dao, "Flashattention-2: Faster attention with better parallelism and work partitioning," *arXiv preprint arXiv:2307.08691*, 2023.

[11] J. Shah, G. Bikshandi, Y. Zhang, V. Thakkar, P. Ramani, and T. Dao, "Flashattention-3: Fast and accurate attention with asynchrony and low-precision," *Advances in Neural Information Processing Systems*, vol. 37, pp. 68 658–68 685, 2024.

[12] A. Gu and T. Dao, "Mamba: Linear-time sequence modeling with selective state spaces," *arXiv preprint arXiv:2312.00752*, 2023.

[13] J. Ho, A. Jain, and P. Abbeel, "Denoising diffusion probabilistic models," *Advances in neural information processing systems*, vol. 33, pp. 6840–6851, 2020.

[14] X. Li, J. Thickstun, I. Gulrajani, P. S. Liang, and T. B. Hashimoto, "Diffusion-lm improves controllable text generation," *Advances in neural information processing systems*, vol. 35, pp. 4328–4343, 2022.

[15] A. Liu, B. Feng, B. Wang, B. Wang, B. Liu, C. Zhao, C. Dengr, C. Ruan, D. Dai, D. Guo *et al.*, "Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model," *arXiv preprint arXiv:2405.04434*, 2024.

[16] B. Sun, Z. Huang, H. Zhao, W. Xiao, X. Zhang, Y. Li, and W. Lin, "Llumnix: Dynamic scheduling for large language model serving," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 173–191.

[17] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[18] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "{TVM}: An automated {End-to-End} optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 578–594.

[19] P. Tillet, H.-T. Kung, and D. Cox, "Triton: an intermediate language and compiler for tiled neural network computations," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2019, pp. 10–19.

[20] M. Wu, X. Cheng, O. Padon, and Z. Jia, "A multi-level superoptimizer for tensor programs," *arXiv preprint arXiv:2405.05751*, 2024.

[21] B. F. Spector, S. Arora, A. Singhal, D. Y. Fu, and C. Ré, "Thunderkittens: Simple, fast, and adorable ai kernels," *arXiv preprint arXiv:2410.20399*, 2024.

[22] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen *et al.*, "Ansor: Generating {High-Performance} tensor programs for deep learning," in *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, 2020, pp. 863–879.

[23] S. Zheng, R. Chen, A. Wei, Y. Jin, Q. Han, L. Lu, B. Wu, X. Li, S. Yan, and Y. 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*, 2022, pp. 874–887.

[24] A. Ouyang, S. Guo, S. Arora, A. L. Zhang, W. Hu, C. Ré, and A. Mirhoseini, "Kernelbench: Can llms write efficient gpu kernels?" *arXiv preprint arXiv:2502.10517*, 2025.

[25] R. T. Lange, A. Prasad, Q. Sun, M. Faldor, Y. Tang, and D. Ha, "The ai cuda engineer: Agentic cuda kernel discovery, optimization and composition," Technical Report, Sakana AI (preprint), Feb. 2025. [Online]. Available: https://pub.sakana.ai/static/paper.pdf

[26] W. Chen, J. Zhu, Q. Fan, Y. Ma, and A. Zou, "Cuda-llm: Llms can write efficient cuda kernels," 2025. [Online]. Available: https://arxiv.org/abs/2506.09092

[27] M. Andrews and S. Witteveen, "Gpu kernel scientist: An llm-driven framework for iterative kernel optimization," 2025. [Online]. Available: https://arxiv.org/abs/2506.20807

[28] C. Baronio, P. Marsella, B. Pan, S. Guo, and S. Alberti, "Kevin: Multi-turn rl for generating cuda kernels," 2025. [Online]. Available: https://arxiv.org/abs/2507.11948

[29] X. Li, X. Sun, A. Wang, J. Li, and C. Shum, "Cuda-l1: Improving cuda optimization via contrastive reinforcement learning," 2025. [Online]. Available: https://arxiv.org/abs/2507.14111

[30] "Kernelbench," https://github.com/ScalingIntelligence/KernelBench, 2025.

[31] L. Zheng, L. Yin, Z. Xie, C. L. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez *et al.*, "Sglang: Efficient execution of structured language model programs," *Advances in Neural Information Processing Systems*, vol. 37, pp. 62 557–62 583, 2025.

[32] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu *et al.*, "Autogen: Enabling next-gen llm applications via multi-agent conversations," in *First Conference on Language Modeling*, 2024.

[33] C.-A. Cheng, A. Nie, and A. Swaminathan, "Trace is the next autodiff: Generative optimization with rich feedback, execution traces, and llms," *Advances in Neural Information Processing Systems*, vol. 37, pp. 71 596–71 642, 2024.

[34] S. Hong, X. Zheng, J. Chen, Y. Cheng, J. Wang, C. Zhang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou *et al.*, "Metagpt: Meta programming for multi-agent collaborative framework," *arXiv preprint arXiv:2308.00352*, vol. 3, no. 4, p. 6, 2023.

[35] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, C. Zhang, J. Wang, Z. Wang, S. K. S. Yau, Z. Lin *et al.*, "Metagpt: Meta programming for a multi-agent collaborative framework." International Conference on Learning Representations, ICLR, 2024.

[36] G. Li, H. Hammoud, H. Itani, D. Khizbullin, and B. Ghanem, "Camel: Communicative agents for" mind" exploration of large language model society," *Advances in Neural Information Processing Systems*, vol. 36, pp. 51 991–52 008, 2023.

[37] D. Huang, J. M. Zhang, M. Luck, Q. Bu, Y. Qing, and H. Cui, "Agentcoder: Multi-agent-based code generation with iterative testing and optimisation," 2024. [Online]. Available: https://arxiv.org/abs/2312.13010

[38] C. Qian, W. Liu, H. Liu, N. Chen, Y. Dang, J. Li, C. Yang, W. Chen, Y. Su, X. Cong, J. Xu, D. Li, Z. Liu, and M. Sun, "Chatdev: Communicative agents for software development," 2024. [Online]. Available: https://arxiv.org/abs/2307.07924

[39] A. Wei, A. Nie, T. S. F. X. Teixeira, R. Yadav, W. Lee, K. Wang, and A. Aiken, "Improving parallel program performance with LLM optimizers via agent-system interfaces," in *Forty-second International Conference on Machine Learning*, 2025. [Online]. Available: https://openreview.net/forum?id=3h80HyStMH

[40] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *Acm Sigplan Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[41] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: A compiler infrastructure for the end of moore's law," 2020. [Online]. Available: https://arxiv.org/abs/2002.11054

[42] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[43] V. Thakkar, P. Ramani, C. Cecka, A. Shivam, H. Lu, E. Yan, J. Kosaian, M. Hoemmen, H. Wu, A. Kerr, M. Nicely, D. Merrill, D. Blasig, F. Qiao, P. Majcher, P. Springer, M. Hohnerbach, J. Wang, and M. Gupta, "Cutlass (cuda templates for linear algebra subroutines)," Version 3.0.0, GitHub, 2023, cUDA Templates for Linear Algebra Subroutines. [Online]. Available: https://github.com/NVIDIA/cutlass/tree/v3.0.0

[44] N. Rotem, J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein *et al.*, "Glow: Graph lowering compiler techniques for neural networks," *arXiv preprint arXiv:1805.00907*, 2018.

[45] R. Wei, L. Schwartz, and V. Adve, "Dlvm: A modern compiler infrastructure for deep learning systems," *arXiv preprint arXiv:1711.03016*, 2017.

[46] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks." *Proceedings of Machine Learning and Systems*, vol. 1, pp. 1–13, 2019.

[47] S. Zheng, R. Chen, Y. Jin, A. Wei, B. Wu, X. Li, S. Yan, and Y. Liang, "Neoflow: A flexible framework for enabling efficient compilation for high performance dnn training," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 11, pp. 3220–3232, 2021.

[48] A. Wei, H. Song, M. Hidayetoglu, E. Slaughter, S. K. Lele, and A. Aiken, "Task-based programming for adaptive mesh refinement in compressible flow simulations," *arXiv preprint arXiv:2508.05020*, 2025.

[49] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[50] J. Taneja, A. Laird, C. Yan, M. Musuvathi, and S. K. Lahiri, "Llm-vectorizer: Llm-based verified loop vectorizer," in *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, 2025, pp. 137–149.

[51] Z. Zheng, K. Wu, L. Cheng, L. Li, R. C. O. Rocha, T. Liu, W. Wei, J. Zeng, X. Zhang, and Y. Gao, "Vectrans: Enhancing compiler auto-vectorization through llm-assisted code transformations," 2025. [Online]. Available: https://arxiv.org/abs/2503.19449

[52] A. Wei, T. Suresh, H. Tan, Y. Xu, G. Singh, K. Wang, and A. Aiken, "Improving assembly code performance with large language models via reinforcement learning," 2025. [Online]. Available: https://arxiv.org/abs/2505.11480

[53] A. Wei, A. Nie, T. S. Teixeira, R. Yadav, W. Lee, K. Wang, and A. Aiken, "Improving parallel program performance through dsl-driven code generation with llm optimizers," *arXiv preprint arXiv:2410.15625*, 2024.

[54] A. Wei, R. Yadav, H. Song, W. Lee, K. Wang, and A. Aiken, "Mapple: A domain-specific language for mapping distributed heterogeneous parallel programs," *arXiv preprint arXiv:2507.17087*, 2025.

[55] Y. Zhai, S. Yang, K. Pan, R. Zhang, S. Liu, C. Liu, Z. Ye, J. Ji, J. Zhao, Y. Zhang *et al.*, "Enabling tensor language model to assist in generating {High-Performance} tensor programs for deep learning," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 289–305.

[56] J. Li, S. Li, Z. Gao, Q. Shi, Y. Li, Z. Wang, J. Huang, H. Wang, J. Wang, X. Han, Z. Liu, and M. Sun, "Tritonbench: Benchmarking large language model capabilities for generating triton operators," 2025. [Online]. Available: https://arxiv.org/abs/2502.14752

[57] T. Chen, B. Xu, and K. Devleker, "Automating gpu kernel generation with deepseek-r1 and inference time scaling," NVIDIA Technical Blog, Feb. 2025. [Online]. Available: https://developer.nvidia.com/blog/automating-gpu-kernel-generation-with-deepseek-r1-and-inference-time-scaling/

[58] A. Wei, J. Cao, R. Li, H. Chen, Y. Zhang, Z. Wang, Y. Sun, Y. Liu, T. S. Teixeira, D. Yang *et al.*, "Equibench: Benchmarking code reasoning capabilities of large language models via equivalence checking," *arXiv e-prints*, pp. arXiv–2502, 2025.

[59] A. Wei, J. Cao, R. Li, H. Chen, Y. Zhang, Z. Wang, Y. Liu, T. S. Teixeira, D. Yang, K. Wang *et al.*, "Equibench: Benchmarking large language models' understanding of program semantics via equivalence checking," *arXiv preprint arXiv:2502.12466*, 2025.

[60] L. A. Agrawal, S. Tan, D. Soylu, N. Ziems, R. Khare, K. Opsahl-Ong, A. Singhvi, H. Shandilya, M. J. Ryan, M. Jiang, C. Potts, K. Sen, A. G. Dimakis, I. Stoica, D. Klein, M. Zaharia, and O. Khattab, "Gepa: Reflective prompt evolution can outperform reinforcement learning," 2025. [Online]. Available: https://arxiv.org/abs/2507.19457

[61] A. Novikov, N. Vũ, M. Eisenberger, E. Dupont, P.-S. Huang, A. Z. Wagner, S. Shirobokov, B. Kozlovskii, F. J. R. Ruiz, A. Mehrabian, M. P. Kumar, A. See, S. Chaudhuri, G. Holland, A. Davies, S. Nowozin, P. Kohli, and M. Balog, "Alphaevolve: A coding agent for scientific and algorithmic discovery," 2025. [Online]. Available: https://arxiv.org/abs/2506.13131

[62] A. Ouyang, P. Liang, and A. Mirhoseini, "Surprisingly fast ai-generated kernels we didn't mean to publish (yet)," Scaling Intelligence Lab blog, Stanford University, 2025. [Online]. Available: https://scalingintelligence.stanford.edu/blogs/fastkernels/

[63] "Openai agents sdk," https://github.com/openai/openai-agents-python, 2025.

[64] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[65] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[66] W. Liang, T. Liu, L. Wright, W. Constable, A. Gu, C.-C. Huang, I. Zhang, W. Feng, H. Huang, J. Wang, S. Purandare, G. Nadathur, and S. Idreos, "Torchtitan: One-stop pytorch native solution for production ready LLM pretraining," in *The Thirteenth International Conference on Learning Representations*, 2025. [Online]. Available: https://openreview.net/forum?id=SFN6Wm7YBI