# FUSIONAGENT: JOINT SEARCH OVER FUSION BOUNDARIES AND KERNEL IMPLEMENTATIONS FOR WHOLE-MODEL INFERENCE

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Kernel fusion can reduce launch and data-movement overhead in model inference, but optimizing a complete model requires jointly deciding which operations to fuse and how to implement each fused region. Larger regions expose more optimization opportunities but may increase resource pressure and reduce GPU parallelism; their implementation spaces are also difficult to explore under a limited budget, particularly when LLM-generated decisions and code are unreliable. We present FUSIONAGENT, an agentic system that addresses these challenges to optimize full-model inference. GRAPHAGENT searches fusion boundaries over multiple phases, while KERNELAGENT implements the proposed regions; only verified, measured improvements rewrite the graph and expose larger regions for subsequent phases. KERNELAGENT uses ECHO search to rank a diverse candidate pool from profiles and search history, decompose plans into independently evaluated units, distinguish an ineffective hypothesis from an inefficient implementation, and reuse experimental outcomes across rounds. Generated kernels are checked against reference values at their graph boundaries and validated again after model integration. Thus, LLMs make the semantic decisions, while deterministic verification and hardware measurements control acceptance. FUSIONAGENT automatically produces optimized models that can be integrated with production inference engines while preserving distributed tensor-parallel execution; we implement and evaluate this integration with SGLang. Across GLM-4.7-358B and LLaDA2.2-Flash workloads on eight NVIDIA H20-3e GPUs, FUSIONAGENT improves end-to-end execution by $1.54$–$1.89\times$ over SGLang v0.5.8 and $1.18$–$1.50\times$ over the more optimized SGLang v0.5.17. On the fused-region benchmarks, KERNELAGENT outperforms `torch.compile` in every case and achieves the best latency on three out of four, with up to a $2.17\times$ advantage over the strongest competing kernel agent.

## 1 INTRODUCTION

Efficient inference is essential for deploying large language models, yet optimizing end-to-end model latency remains a labor-intensive systems task. Modern inference frameworks provide highly optimized kernels and libraries for common operations, but executing a model as a sequence of separate operators incurs repeated launch overhead and materializes intermediate tensors in global memory. Kernel fusion reduces these costs by combining neighboring operations, keeping intermediate values closer to the processor, and jointly scheduling their computation. Traditional AI compilers such as TVM and `torch.compile` capture computation graphs and apply operator fusion and backend code generation (Chen et al., 2018; Ansel et al., 2024). Their fusion opportunities, however, are constrained by predefined transformations and the operator combinations supported by their backends, leaving model- and deployment-specific opportunities outside the encoded search space.

Recent advances in large language models offer a new path toward automating this optimization process. Given a predefined computational unit, LLM-based agents can generate implementations, interpret compiler and profiler feedback, and iteratively search for faster code (Ouyang et al., 2025; Zhang et al., 2025; 2026; Dai et al., 2026). Most kernel agents nevertheless optimize tasks whose



Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

boundaries are specified in advance, so they automate implementation search without deciding which portions of a model should be optimized together. Moreover, kernel optimization is often evaluated in isolated benchmark harnesses whose interfaces and execution environments differ from production inference systems, leaving additional integration work before optimized kernels can be deployed in end-to-end model execution (Oliaro et al., 2026). LEGO extends agentic synthesis to a full model by recursively decomposing its source-level module hierarchy and applying bottom-up fusion to optimized subproblems (Xu et al., 2026). This source-guided organization makes the search tractable, but it does not directly search arbitrary operator-level partitions of the computation graph.

At the other extreme, megakernel systems place most or all computations of an inference step in one persistent kernel (Li et al., 2026; Cheng et al., 2026b). However, more fusion is not always better. As a fused region grows, each CTA may consume more registers and shared memory, reducing occupancy and the number of CTAs that execute in parallel; synchronization, load imbalance, and incompatible scheduling requirements can add further costs. Effective fusion must therefore choose boundaries that balance launch and data-movement savings against resource pressure and available parallelism, and it must find an efficient implementation within each chosen region.

We present FUSIONAGENT, an agentic system that uses kernel fusion to optimize full-model inference. Its design addresses three challenges. First, choosing a fusion boundary requires balancing reductions in kernel-launch and data-movement overhead against the resource pressure and loss of parallelism that can accompany a larger kernel. GRAPHAGENT therefore searches boundaries over multiple phases, while KERNELAGENT implements each proposed region; only measured improvements rewrite the graph and expose larger opportunities for the next phase. Second, the implementation space is too large to explore exhaustively, and unreliable LLMs may both pursue unpromising ideas and produce inefficient code for sound ones. KERNELAGENT addresses this with ECHO search, which ranks a diverse candidate pool using profiles and search history, decomposes plans into independently tested units to attribute poor results to either an idea or its implementation, and reuses accumulated outcomes across rounds. Third, generated kernels must preserve the behavior of the full model. FUSIONAGENT checks each replacement against reference values captured at its graph boundary and validates the rewritten model again after integration. FUSIONAGENT deploys LLMs to make almost all semantic decisions throughout this whole process, while deterministic correctness checks and hardware measurements control which kernels enter the model.

FUSIONAGENT automatically emits an optimized model implementation that can integrate with production inference engines, such as SGLang (Zheng et al., 2024), and supports distributed tensor-parallel execution. We evaluate FUSIONAGENT with production-scale GLM and LLaDA models on eight NVIDIA H20-3e GPUs. The evolution of SGLang illustrates an important optimization gap: v0.5.17 substantially improves the popular GLM model through manual fusion and faster GEMM and MoE implementations, but only improves the LLaDA model by about $3\%$. Over this newer SGLang, FUSIONAGENT improves LLaDA by approximately $50\%$, demonstrating its value for newly added models that have not yet accumulated extensive manual tuning in SGLang; it still further improves the well-tuned GLM model by approximately $20\%$, indicating that profitable fusion opportunities remain even after substantial manual kernel-fusion optimization. Relative to SGLang v0.5.8, it improves GLM by approximately $90\%$ for batch-size-one decoding and $50\%$ at batch size 32. Our ablations further show that multiple graph-fusion phases reduce kernel launches from 51 to 13 on GLM and from 41 to 16 on LLaDA. Across fused-region benchmarks, KERNELAGENT outperforms `torch.compile` in every case and achieves the best latency on three of four regions, with up to a $2.17\times$ advantage over the strongest competing kernel agent.

This paper makes the following contributions:

- We build an end-to-end agentic optimization system for accelerating new models in production inference engines. FUSIONAGENT automatically produces optimized, verified model implementations that can preserve distributed tensor-parallel execution. We demonstrate this capability through an integration with SGLang, optimizing LLaDA on eight GPUs without extensive model-specific kernel engineering.
- We introduce a joint, progressive optimization procedure that iteratively invokes GRAPH AGENT and KERNELAGENT to search fusion boundaries and optimize increasingly large and complex fused kernels. This alternating process fuses many original kernels into efficient large fused kernels and yields substantial end-to-end inference speedups.

- We introduce ECHO search to explore the large optimization spaces of fused regions under a limited evaluation budget and unreliable LLM decisions. We demonstrate that this approach can generate much more efficient fused kernels than `torch.compile` and recent kernel agents.

## 2 RELATED WORK

**LLM-based kernel generation and optimization.** KernelBench benchmarks correct and efficient kernel generation, including multi-operator fusion workloads (Ouyang et al., 2025). CudaForge, AccelOpt, and CUDA Agent combine LLM generation with compiler feedback, hardware profiling, candidate search, memory, or reinforcement learning (Zhang et al., 2025; 2026; Dai et al., 2026). The KernelAgent framework combines profile-guided diagnosis, beam search, parallel optimization, and reflective memory (Cheng et al., 2026a). Atrex-Kernel-Agent targets operators sampled from production traces using a profile-driven measure–revise loop, optimization dropout, and a layered GPU-optimization knowledge base (Yang et al., 2026). Kernel Design Agents (KDA) uses KernelWiki, a structured knowledge base of GPU architectures, optimization techniques, and kernel patterns, in an iterative CUDA optimization workflow (MIT HAN Lab, 2026a;b). These systems primarily optimize regions whose boundaries are given in advance. Unlike their typical use of standalone benchmark harnesses, FUSIONAGENT is designed for integration with production inference engines: it captures computation graphs from deployed models, installs verified replacements into the runtime, and produces executable tensor-parallel models without manual kernel integration. We instantiate this design in SGLang. Its KERNELAGENT also addresses the larger implementation spaces created by graph-level fusion through multi-lineage search, plan decomposition, and cross-round memory.

**Agentic graph fusion and megakernel generation.** KernelFalcon partitions programs into fusion groups and generates Triton implementations, while FACT realizes discovered graph patterns with auto-tuned CUTLASS kernels (Wang et al., 2025; Heidari & Nikolopoulos, 2026). ForgeMegakernel instead places an autoregressive decode step in a persistent megakernel and uses intermediate-state oracles to guide its implementation (Li et al., 2026). The closest work, LEGO, performs bottom-up full or partial fusion over a recursive AND–OR FusionTree derived from the source-level module hierarchy (Xu et al., 2026). FUSIONAGENT instead searches an operator-level graph and repeatedly installs profitable kernels as new graph nodes, allowing later rounds to combine operators and generated kernels across module boundaries. Coupled with KERNELAGENT, FUSIONAGENT produces substantially more aggressive fusion than the results reported by LEGO.

**Computation-graph and tensor compilers.** TVM, TASO, and `torch.compile` optimize graphs through compiler-defined rewrites, fusion, cost models, and tensor-program generation (Chen et al., 2018; Jia et al., 2019; Ansel et al., 2024). Ansor and Welder add schedule search and tile-level fusion, while Mirage jointly searches algebraic transformations and tensor programs (Zheng et al., 2020; Shi et al., 2023; Wu et al., 2025). Mirage Persistent Kernel further lowers multi-GPU inference into an SM-level task graph executed by a persistent megakernel (Cheng et al., 2026b). Their candidate spaces remain encoded in compiler rules, intermediate representations, and code-generation backends. FUSIONAGENT instead uses LLMs to propose workload-specific regions and implementations, with deterministic verification and direct measurement controlling acceptance.

## 3 DESIGN

FUSIONAGENT is an end-to-end agentic system to accelerate large language model inference in production inference engines. Given a model and an inference configuration, it captures the runtime computation graphs, optimizes selected regions, and emits an executable implementation for the inference engine. Its primary transformation is kernel fusion: rather than optimizing each original operator independently, FUSIONAGENT groups operations into regions that may cross their original boundaries and implements each region with one or more fused GPU kernels. Fusion exposes opportunities to eliminate intermediate memory traffic and launches, reuse data on faster memory, and jointly schedule computations. A complete overview of the optimization procedure is shown in Figure 1.

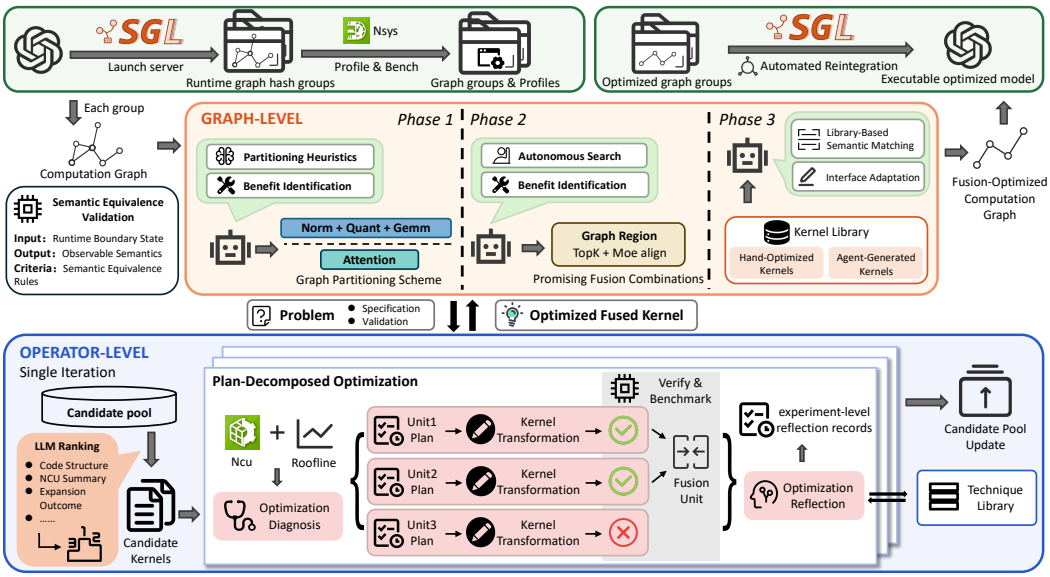



Figure 1: Overview of FUSIONAGENT. Starting from a model deployed in SGLang, FUSIONAGENT captures its runtime computation graphs and organizes layer-level graphs into structurally equivalent groups. GRAPHAGENT progressively selects fusion regions and rewrites the graph using measured implementation outcomes, while KERNELAGENT performs evidence-guided, plan-factored search within each selected region. Optimized implementations are then integrated back into SGLang for end-to-end execution.

### 3.1 GRAPH-LEVEL OPTIMIZATION

Graph-level optimization must determine how far each fused region should extend. A larger region can eliminate more kernel launches and intermediate memory traffic and enable joint scheduling, but it can also increase register and shared-memory usage, reduce occupancy, and limit GPU parallelism. The best boundary therefore cannot be determined from graph structure alone and must be evaluated through an implemented kernel. Each profitable implementation then replaces its region in the graph, creating new opportunities to expand or reorganize neighboring regions. GRAPHAGENT consequently searches fusion boundaries over multiple phases, using measurements from earlier phases to guide decisions on the evolving graph.

The first two phases jointly evolve fusion boundaries and their implementations. GRAPHAGENT proposes non-overlapping regions, and KERNELAGENT implements them. A generated implementation is admitted only after it passes correctness verification and improves measured performance; GRAPHAGENT then replaces its region and searches the rewritten graph in the next phase. A later region can thus contain both original operators and fused kernels produced earlier, enabling progressively larger fusions. Figure 1 illustrates this process, and Algorithm 2 in Appendix A.1 provides the detailed procedure. The first two phases restrict fusion to operators supported by KERNELAGENT's code-generation backend. GRAPHAGENT marks unsupported operators as excluded nodes and uses them to split the graph into supported subgraphs, so proposed regions cannot cross these nodes. In the partition phase, the LLM divides each supported subgraph into non-overlapping regions using execution and dataflow compatibility, fusion affinity, and potential savings in launches and intermediate materialization. In the exploration phase, it uses fusion history and concrete implementations in the rewritten graph to expand or recombine regions beyond the initial partition.

The final phase relaxes the code-generation backend constraint by considering replacements from a library of human-optimized kernels. Because KERNELAGENT does not generate these implementations, unsupported operators no longer act as barriers and GRAPHAGENT may consider regions spanning any operator type. A matched replacement must still pass the same correctness and performance checks before it is installed in the graph.

### 3.2 OPERATOR-LEVEL OPTIMIZATION

After GRAPHAGENT selects a fusion region, KERNELAGENT must address two challenges. First, a fused region has a large implementation space, but only a small fraction of its candidates can be evaluated. The search must prioritize promising candidates without prematurely discarding slower ones that may enable subsequent optimizations. Second, LLMs make the major semantic decisions in this search, including ranking candidates, diagnosing bottlenecks, and generating plans and implementations, but these decisions are unreliable. In particular, inefficient generated code can make a sound optimization hypothesis appear ineffective, so a poor result should not automatically eliminate its candidate or idea.

To address both challenges, KERNELAGENT performs *ECHO search*: Exploration-aware Candidate ranking, Hypothesis attribution, and Outcome memory. ECHO maintains a bounded pool of runnable, verified, and measured implementations. It ranks candidates using both performance and exploration evidence, decomposes plans to attribute outcomes to individual hypotheses, and records the resulting evidence for reuse across rounds. Selecting multiple candidates limits the effect of an incorrect LLM ranking, while deterministic verification and hardware measurement decide which generated implementations enter the pool. Algorithm 4 in Appendix A.2 presents the complete procedure.

**Exploration-aware candidate ranking.** Selecting candidates solely by runtime tends to overexploit the current fastest implementation. A slightly slower candidate may introduce a useful structural change, expose a clear hardware bottleneck, or remain insufficiently explored. Conversely, a fast candidate that has already been expanded repeatedly without yielding new evidence may not warrant additional search effort. KERNELAGENT therefore separates admission to the candidate pool from selection for further expansion.

At the beginning of each round, an LLM ranks compact candidate summaries using runtime, hardware profiles, search depth, expansion count, and previous outcomes. It selects multiple parent candidates, balancing continued refinement of strong implementations with exploration of alternative branches. The pool remains bounded through code deduplication and a runtime-based capacity limit. A new implementation need not immediately outperform its parent: if its slowdown remains within a predefined bound, it may be retained as a starting point for subsequent optimizations. This design preserves potentially useful alternatives while keeping the search state tractable.

**Hypothesis attribution through plan decomposition.** For each parent candidate, KERNEL AGENT uses its code, profiles, timing breakdown, and previous outcomes to formulate hypotheses about specific performance improvement mechanisms, such as increasing parallelism, reducing memory traffic, register pressure or kernel launches. It translates each hypothesis into a plan and decomposes independently testable changes into separate units. Every unit is implemented from the same parent, and is verified and benchmarked independently.

KERNELAGENT then checks whether each implementation realizes its predicted mechanism. A realized mechanism that improves runtime supports the hypothesis, while one that regresses provides evidence against it in the current context. If the generated code fails to realize the mechanism, the failure is attributed to the implementation and the hypothesis may be retried; results within measurement noise remain inconclusive. Independently successful units are combined and re-evaluated, and the combined candidate is retained only when it improves on its constituents. This attribution prevents an inefficient implementation from prematurely eliminating a promising optimization idea. Algorithm 5 illustrates this design; further details are provided in Appendix A.2.

**Outcome memory across rounds.** KERNELAGENT maintains two complementary forms of optimization memory: experiment-level reflection records and a reusable Technique Library. After each evaluated implementation, the agent records the region context, optimization hypothesis, code change, verification and profiling results, whether the intended mechanism was realized, and the resulting attribution. These records retain successful, failed, and inconclusive trials, providing evidence about what has been attempted and why an optimization did or did not work. A subset of outcomes is further distilled into the Technique Library, which captures reusable implementation mechanisms and harmful patterns that generalize beyond a single candidate or fusion region.

Before ranking candidates and generating new hypotheses, KERNELAGENT retrieves relevant reflection records and applicable library entries. Reflection records help avoid repeated failures, revisit promising hypotheses whose previous implementations were ineffective, and build on successful candidates, while the Technique Library transfers reusable implementation principles across rounds and related regions. Frequently useful techniques can be further consolidated into a static technique set, enabling broadly applicable optimization knowledge to persist across fusion tasks.

**Correctness verification.** A fused implementation replaces a region containing multiple original operations, so checking only the final model output may conceal errors inside the region. The first decision is therefore to use the original model as a reference at the fusion boundary: KERNELAGENT records the region's inputs and corresponding intermediate outputs, then compares a generated implementation with these values. It exercises supported shapes and data types with representative and numerically sensitive inputs, using data-type-appropriate tolerances and checks for invalid values. Compilation and numerical comparison are deterministic. The LLM may propose a repair after a failure, but the repaired kernel must pass the same checks; otherwise it is discarded. After each replacement, GRAPHAGENT executes the rewritten model and checks its outputs again. Local verification catches errors within a fused region, while validation of the rewritten and fully integrated models checks that region interfaces and runtime installation preserve the complete computation.

### 3.3 PRODUCTION INFERENCE ENGINE INTEGRATION

FUSIONAGENT is designed to optimize models within production inference engines rather than export kernels to an isolated benchmark harness. An engine integration captures runtime computation graphs, groups structurally equivalent layers, and optimizes one representative per group while retaining reference tensors for correctness validation. Compatible results can then be reused across graph groups. Our implementation obtains these graphs through SGLang's unified decode-layer interface.

For each representative layer, FUSIONAGENT installs accepted kernels as custom operators in the optimized graph. During serving, the integration matches runtime graphs to optimized structures and replaces the corresponding regions while preserving parameter ownership, distributed communication, and the engine's surrounding execution path. It then validates and emits an executable model without manual kernel porting. We use this procedure to integrate FUSIONAGENT with SGLang and preserve its tensor-parallel execution in our evaluation.

## 4 EVALUATION

We evaluate FUSIONAGENT on two complementary model architectures: GLM-4.7-358B FP8 model (Z.ai, 2025), a production-scale autoregressive mixture-of-experts model with 358 billion parameters, and LLaDA2.2-Flash FP8 model (Bie et al., 2026), a diffusion language model with 100 billion parameters that decodes multiple tokens in parallel. For GLM, we evaluate batch sizes of 1 and 32. For LLaDA, the workload uses batch size 1 with a block of 32 tokens processed in parallel, denoted as B=1/T=32 in Table 1. All experiments use an input length of 1,024 tokens and run with tensor parallelism of 8 on eight NVIDIA H20-3e GPUs. All compared systems use identical model weights, input shapes, and parallelism configurations.

**Baselines and measurement.** We use SGLang as the inference engine and compare against SGLang v0.5.8 and v0.5.17 (SGLang Team, 2026b;a). Under our configuration, v0.5.8 accelerates model execution with `torch.compile` and CUDA Graphs, while v0.5.17 additionally incorporates manually optimized fused operators and faster GEMM and MoE implementations. All configurations use the FA3 attention backend (Shah et al., 2024) and Triton (Tillet et al., 2019) MoE backend, with Triton operator configurations selected using SGLang's built-in tuning scripts. When benchmarking the performance of kernels, we use NVIDIA Nsight Systems to collect execution traces and measure the latency of each kernel execution. This can give us the most accurate timing measurement and avoid any kernel launch overhead when measuring the time for small kernels.

**Evaluation Protocol.** When comparing KERNELAGENT with other kernel agents, all agentic methods use the same frontier LLM as the underlying LLM. For kernel optimization, each fused

Table 1: End-to-end latency for one model forward and the latency of selected layers. Speedups use the indicated SGLang baseline; #K denotes the number of kernel launches.

**(a) The runtime of one model forward.**

| | GLM, BS=1 | GLM, BS=32 | LLaDA, B=1/T=32 |
|---|---|---|---|
| SGLang 0.5.8 (ms) | 14.866 | 22.588 | 6.678 |
| SGLang 0.5.17 (ms) | 9.472 | 17.312 | 6.493 |
| FusionAgent (ms) | **7.850** | **14.657** | **4.330** |
| Speedup vs. 0.5.8 | **1.89**× | **1.54**× | **1.54**× |
| Speedup vs. 0.5.17 | **1.21**× | **1.18**× | **1.50**× |

**(b) The latency of a Transformer layer ($\mu$s) and kernel count.**

| | | SGLang 0.5.8 | | SGLang 0.5.17 | | FusionAgent | |
|---|---|---|---|---|---|---|---|
| Workload | Layer | Time | #K | Time | #K | Time | #K |
| GLM, BS=1 | Layer 1 (Dense) | 158.46 | 21 | 84.64 | 16 | **72.56** | **11** |
| | Layer 3 (Sparse) | 164.51 | 37 | 109.17 | 21 | **82.45** | **14** |
| GLM, BS=32 | Layer 1 (Dense) | 244.29 | 22 | 118.77 | 17 | **117.50** | **14** |
| | Layer 3 (Sparse) | 251.34 | 38 | 190.69 | 24 | **134.71** | **13** |
| LLaDA, B=1/T=32 | Layer 3 (Sparse) | 188.33 | 36 | 202.83 | 37 | **118.26** | **16** |

region is optimized for up to five rounds, with a budget of nine implementation expansions per round. We report the lowest-latency verified candidate found within this budget.

### 4.1 Major experiments

Table 1(a) shows that FusionAgent improves all evaluated workloads. Its largest gain occurs for batch-size-one autoregressive decoding, where limited work per operation makes kernel launches and intermediate materialization a large fraction of latency. The relative gain decreases as GLM's batch size grows and its GEMM and MoE kernels account for more execution time, but remains approximately $50\%$ over SGLang v0.5.8. Upgrading SGLang from v0.5.8 to v0.5.17 substantially improves GLM through additional manual fusion and faster GEMM and MoE implementations, illustrating how popular models accumulate model-specific optimizations as an inference engine evolves. The same upgrade changes LLaDA latency by only about $3\%$, leaving much of the newer model's optimization potential unexplored. FusionAgent fills this gap automatically: it accelerates LLaDA by $50\%$ over v0.5.17 while also improving the already well-tuned GLM workloads by approximately $20\%$. These results position FusionAgent as a practical tool for accelerating new SGLang models before comparable manual optimization effort becomes available.

Table 1(b) attributes these end-to-end gains to lower layer latency and fewer kernel launches. By searching the full computation graph of a representative Transformer layer, FusionAgent fuses operations around major compute kernels and crosses boundaries left by compiler-generated and manually written kernels. The effect is most pronounced for LLaDA's sparse Transformer layer, which is $1.72\times$ faster than SGLang v0.5.17 while reducing the number of launches from 37 to 16.

### 4.2 Ablation studies

#### 4.2.1 Effectiveness of Progressive Graph Fusion

Table 2 evaluates the progressive graph-fusion procedure by reporting the best layer implementation after each phase. The first phase captures the most direct opportunities and produces most of the improvement for the simpler dense layers, after which their latency and kernel count remain unchanged. This stabilization is desirable: once no profitable expansion is found, GraphAgent retains the existing implementation rather than forcing a larger fusion.

Later phases remain valuable for the more complex sparse layers. Each accepted region implementation rewrites the graph as a new node, enabling GraphAgent to propose a larger region and allowing KernelAgent to reuse the optimized implementation from the preceding phase. Con-

Table 2: Per-layer performance across successive graph-fusion phases. The Initial stage denotes the unoptimized starting graph from SGLang v0.5.8 with `torch.compile` disabled. Each entry reports latency in $\mu$s, with the number of kernel launches in parentheses. Appendix B.1 provides a representative case study on the third layer of LLaDA, illustrating the intermediate graph states and accepted rewrites produced by the automated graph-fusion process.

| Workload | Layer | Initial | P1: Partition | P2: Exploration | P3: Reuse |
|---|---|---|---|---|---|
| GLM, BS=1 | Layer 1 (Dense) | 127.95 (21) | 72.56 (11) | 72.56 (11) | 72.56 (11) |
| | Layer 3 (Sparse) | 197.22 (48) | 83.09 (15) | 82.45 (14) | 82.45 (14) |
| GLM, BS=32 | Layer 1 (Dense) | 171.47 (22) | 117.50 (14) | 117.50 (14) | 117.50 (14) |
| | Layer 3 (Sparse) | 327.97 (51) | 164.24 (17) | 157.36 (15) | 134.71 (13) |
| LLaDA, B=1/T=32 | Layer 3 (Sparse) | 270.02 (41) | 184.22 (25) | 149.84 (18) | 118.26 (16) |

Table 3: Latency of four operator regions measured from NVIDIA Nsight Systems traces ($\mu$s). For KDA, we report the fastest result from three independent runs. Detailed operator semantics, tensor shapes, and workload configurations are provided in the appendix B.2.

| Method | MoE alignment | Quant + FP8 O-proj GEMM | MoE gate + top-k + align | Add + RMSNorm + QKV GEMM + QK norm + RoPE |
|---|---|---|---|---|
| `torch.compile` | 27.82 | 25.79 | 82.18 | 45.30 |
| KernelAgent (Meta) | 12.40 | 6.08 | 117.09 | 20.19 |
| Atrex Kernel Agent (AKA) | 6.08 | 8.58 | 68.93 | 57.26 |
| Kernel Design Agents (KDA) | **2.05** | 4.83 | 19.54 | 12.86 |
| Our KERNELAGENT | 2.67 | **3.53** | **9.02** | **7.58** |

sequently, the GLM batch-size-32 sparse layer continues to improve through all three phases, while its kernel count falls from 51 to 13. LLaDA shows an even clearer progression: successive phases reduce its sparse Transformer layer from 41 to 16 launches and improve latency at every step. These results show that iterative rewriting exposes profitable fusion opportunities that are unavailable to a single-pass partition, particularly for computation graphs with complex MoE structure.

#### 4.2.2 EFFECTIVENESS OF OPERATOR-LEVEL OPTIMIZATIONS

We compare KERNELAGENT against `torch.compile`, Meta KernelAgent, Atrex Kernel Agent (AKA), and Kernel Design Agents (KDA) on the same extracted fused regions. `torch.compile` serves as the compiler baseline, while the other three methods represent recent agentic kernel-optimization workflows. All methods except KDA follow the evaluation protocol described above. For KDA, we follow its native organization with nine optimization directions and up to five attempts per direction. Its optimization process is orchestrated through skill-based guidance rather than a fixed deterministic sequence of optimization actions, which leads to noticeable run-to-run variation in the explored implementations. We therefore run KDA three times per region and report the fastest verified result, giving KDA the benefit of this variability. Within the allocated optimization budgets, all methods reach a stable best candidate.

Table 3 first shows that kernel agents are generally effective: they outperform `torch.compile` on most of the evaluated regions. Their effectiveness varies across tasks and runs, however, and some agents regress on complex fused regions. KDA is particularly unstable across runs, so we give it a favorable evaluation by running it three times on each region and reporting its fastest result.

Even under this protocol, KERNELAGENT achieves the lowest latency on three of the four regions and consistently outperforms `torch.compile`. It loses only on the relatively narrow MoE alignment region, where KDA is faster by a small absolute margin. KERNELAGENT has a clearer advantage on the larger heterogeneous regions, outperforming the strongest competing result by $2.17\times$ for MoE gate, top-$k$, and alignment, and by $1.70\times$ for the region combining normalization, projection, QK normalization, and rotary embedding. These results show that KERNELAGENT is especially effective at exploring the large implementation spaces created by fusing computations with different data-access and scheduling requirements.

Table 4: Ablation of three ECHO mechanisms. Operator-region latency is measured from NVIDIA Nsight Systems traces ($\mu$s). Percentages show the latency increase relative to the full method; a dash indicates an unmeasured setting.

| Configuration | MoE alignment | Quant + FP8 O-proj GEMM | MoE gate + top-k + align | Add + RMSNorm + QKV GEMM + QK norm + RoPE |
|---|---|---|---|---|
| w/o Exploration-aware ranking | 6.85 (+156.6%) | 5.76 (+63.2%) | 13.12 (+45.5%) | 8.19 (+8.0%) |
| w/o Plan decomposition | 6.78 (+153.9%) | 4.96 (+40.5%) | 15.70 (+74.1%) | 9.22 (+21.6%) |
| w/o Technique Library | 11.52 (+331.5%) | 4.22 (+19.5%) | 14.80 (+64.1%) | 8.99 (+18.6%) |
| Full KERNELAGENT | **2.67** | **3.53** | **9.02** | **7.58** |

### 4.2.3 EFFECTIVENESS OF ECHO MECHANISMS

Table 4 shows that all three tested ECHO mechanisms contribute to kernel optimization. Removing exploration-aware ranking, plan decomposition, or the Technique Library increases latency on every evaluated region, although their relative importance varies across workloads. This variation is expected because the mechanisms address different failure modes in the search rather than applying the same optimization mechanism.

Without exploration-aware ranking, the search tends to overexploit currently fast candidates and loses potentially useful alternative lineages. Without hypothesis attribution, a poor generated implementation can incorrectly discredit an otherwise effective optimization idea, reducing the agent's ability to recover from code-generation errors. Without the Technique Library, previously validated optimization mechanisms must be rediscovered instead of being reused across rounds and related regions. The consistent degradation under all three ablations indicates that candidate diversity, mechanism-level attribution, and cross-round reuse provide complementary support for searching large fused-kernel implementation spaces under a limited evaluation budget. Detailed case studies illustrating these failure modes are provided in the appendix B.3.

## 5 CONCLUSION

We presented FUSIONAGENT, an agentic system for optimizing full-model inference by jointly searching fusion boundaries and fused-kernel implementations. GRAPHAGENT progressively rewrites an operator-level computation graph using verified performance improvements, allowing later rounds to discover increasingly aggressive fusion regions. KERNELAGENT uses ECHO search to allocate a limited evaluation budget across promising implementation lineages, distinguish ineffective hypotheses from inefficient implementations, and accumulate optimization evidence across kernel search and graph-fusion rounds. Although LLMs make the major semantic decisions, deterministic correctness checks and direct hardware measurements control which generated kernels enter the model.

We integrate FUSIONAGENT with SGLang and evaluate it on GLM-4.7-358B and LLaDA2.2-Flash using eight NVIDIA H20 GPUs. FUSIONAGENT improves end-to-end model execution by $1.54$–$1.89\times$ over SGLang v0.5.8 and $1.18$–$1.50\times$ over the more optimized SGLang v0.5.17. The comparison between SGLang versions highlights a practical use case: engine upgrades substantially improve the popular GLM model but provide little improvement for the newer LLaDA model, whereas FUSIONAGENT further accelerates LLaDA by $1.50\times$ on the new SGLang version. FUSIONAGENT can therefore optimize newly added models before comparable manual kernel engineering becomes available. Our ablations demonstrate that both progressive graph fusion and ECHO contribute to these gains. On fused-region benchmarks, KERNELAGENT consistently outperforms `torch.compile` and achieves the lowest latency on three of four regions when compared with state-of-the-art kernel agents. Our current implementation generates Triton kernels, which can limit optimization of highly complex fused computations and, in turn, constrain the regions that GRAPH AGENT can fuse profitably. Supporting CUDA and lower-level GPU programming abstractions is a promising direction for extending the search to larger and more heterogeneous regions.

### AI USE STATEMENT

Generative AI is an integral component of FUSIONAGENT, where it is used for candidate ranking, optimization diagnosis and planning, and kernel implementation. Generative AI tools were also used to aid and polish manuscript writing and for retrieval and discovery of related work. All AI-assisted outputs were reviewed by the authors. Generated code used in experiments was executed and validated through correctness checks and empirical measurements, and the authors verified the reported results, claims, and citations. The authors take responsibility for the final content of this work, including text, code, claims, and other artifacts produced with the aid of generative AI.

### REPRODUCIBILITY STATEMENT

We provide the algorithmic definitions, implementation details, experimental protocols, and additional analyses used in this work. The paper and appendix specify the optimization pipeline, workload and hardware configurations, correctness verification procedures, measurement methodology, and evaluation settings. These details are intended to facilitate understanding and independent evaluation of the proposed method and reported system measurements.

## REFERENCES

Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Michael Suo, Phil Tillet, Eikan Wang, Xiaodong Wang, William Wen, Shunting Zhang, Xu Zhao, Keren Zhou, Richard Zou, Ajit Mathews, Gregory Chanan, Peng Wu, and Soumith Chintala. PyTorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 929–947. ACM, 2024. doi: 10.1145/3620665.3640366.

Tiwei Bie, Jiansheng Cai, Maosong Cao, Xiang Cao, Binbin Chen, Fuyuan Chen, Kun Chen, Yuqi Ding, Lun Du, Zhuocheng Gong, Yanmei Gu, Kaiyuan Guan, Xiaoxu Guo, Kairong Han, Zenan Huang, Zijin Huang, Zhenzhong Lan, Chengxi Li, Jianguo Li, Zhe Li, Ruiqi Liang, Xinyao Lin, Huabin Liu, et al. LLaDA2.2: Enabling agentic diffusion language models via levenshtein editing. Technical report, Inclusion AI, 2026. URL https://github.com/inclusionAI/LLaDA2.X/blob/main/LLaDA2_2_tech_report.pdf.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578–594. USENIX Association, 2018. URL https://www.usenix.org/conference/osdi18/presentation/chen.

Kaiming Cheng, Laura Wang, Jack Khuu, Mark Saroufim, Wenyuan Chi, Jiannan Wang, and Joe Isaacson. KernelAgent: Hardware-guided GPU kernel optimization via multi-agent orchestration. PyTorch Blog, March 2026a. URL https://pytorch.org/blog/kernelagent-hardware-guided-gpu-kernel-optimization-via-multi-agent-orchestration/.

Xinhao Cheng, Zhihao Zhang, Yu Zhou, Jianan Ji, Jinchen Jiang, Zepeng Zhao, Ziruo Xiao, Zihao Ye, Yingyi Huang, Ruihang Lai, Hongyi Jin, Bohan Hou, Mengdi Wu, Yixin Dong, Anthony Yip, Zihao Ye, Songting Wang, Wenqin Yang, Xupeng Miao, Tianqi Chen, and Zhihao Jia. MPK: A compiler and runtime for mega-kernelizing tensor programs. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*, pp. 1909–1926. USENIX Association, 2026b. URL https://www.usenix.org/conference/osdi26/presentation/cheng.

Weinan Dai, Hanlin Wu, Qiying Yu, Huan-ang Gao, Jiahao Li, Chengquan Jiang, Weiqiang Lou, Yufan Song, Hongli Yu, Jiaze Chen, Wei-Ying Ma, Ya-Qin Zhang, Jingjing Liu, Mingxuan Wang, Xin Liu, and Hao Zhou. CUDA Agent: Large-scale agentic RL for high-performance CUDA kernel generation. arXiv preprint arXiv:2602.24286, 2026. URL https://arxiv.org/abs/2602.24286.

Sina Heidari and Dimitrios S. Nikolopoulos. FACT: Compositional kernel synthesis with a three-stage agentic workflow. arXiv preprint arXiv:2604.26666, 2026. URL https://arxiv.org/abs/2604.26666.

Zhihao Jia, Oded Padon, James J. Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: Optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 47–62. ACM, 2019. doi: 10.1145/3341301.3359630.

Leshan Li, Zhui Zhu, Xianglong Deng, Yaojian Chen, Qingfeng He, Yuxuan Li, Rong Zhao, Xu Han, and Zhiyuan Liu. ForgeMegakernel: A general framework for efficient auto-regressive model decode megakernels. arXiv preprint arXiv:2609.12379, 2026. URL https://arxiv.org/abs/2609.12379.

MIT HAN Lab. Kernel Design Agents. GitHub repository, 2026a. URL https://github.com/mit-han-lab/kernel-design-agents.

MIT HAN Lab. KernelWiki: Blackwell and hopper kernel optimization knowledge base. GitHub repository, 2026b. URL https://github.com/mit-han-lab/KernelWiki.

Gabriele Oliaro, Yichao Fu, May Jiang, Owen Lu, Junli Wang, Zhihao Jia, Hao Zhang, and Samyam Rajbhandari. FastKernels: Benchmarking GPU kernel generation in production. arXiv preprint arXiv:2605.23215, 2026. URL https://arxiv.org/abs/2605.23215.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. KernelBench: Can LLMs write efficient GPU kernels? In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 47356–47415. PMLR, 2025. URL https://proceedings.mlr.press/v267/ouyang25a.html.

SGLang Team. SGLang v0.5.17. GitHub release, August 2026a. URL https://github.com/sgl-project/sglang/releases/tag/v0.5.17. Commit 2948168.

SGLang Team. SGLang v0.5.8. GitHub release, January 2026b. URL https://github.com/sgl-project/sglang/releases/tag/v0.5.8. Commit 0189f41.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. FlashAttention-3: Fast and accurate attention with asynchrony and low-precision. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi: 10.52202/079017-2193. URL https://papers.nips.cc/paper_files/paper/2024/hash/7ede97c3e082c6df10a8d6103a2eebd2-Abstract-Conference.html.

Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou. Welder: Scheduling deep learning memory access via tile-graph. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pp. 701–718. USENIX Association, 2023. URL https://www.usenix.org/conference/osdi23/presentation/shi.

Philippe Tillet, H. T. Kung, and David 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 (MAPL '19)*, pp. 10–19. ACM, 2019. doi: 10.1145/3315508.3329973.

Laura Wang et al. Kernelfalcon: Deep agent architecture for autonomous GPU kernel generation. PyTorch Blog, November 2025. URL https://pytorch.org/blog/kernelfalcon-autonomous-gpu-kernel-generation-via-deep-agents/.

Mengdi Wu, Xinhao Cheng, Shengyu Liu, Chunan Shi, Jianan Ji, Man Kit Ao, Praveen Velliengiri, Xupeng Miao, Oded Padon, and Zhihao Jia. Mirage: A multi-level superoptimizer for tensor programs. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pp. 21–38. USENIX Association, 2025. URL https://www.usenix.org/conference/osdi25/presentation/wu-mengdi.

Ruiyuan Xu, Shuoming Zhang, Guangli Li, Qiuchu Yu, Rui Zhang, Yangyu Zhang, Hao Qian, Chunwei Xia, Jiacheng Zhao, Chenxi Wang, Xiaobing Feng, Jingling Xue, and Huimin Cui. LEGO: An LLM-enabled hierarchical optimizer for tensor computation graphs with structure-aware search and compositional synthesis. In *Proceedings of the 43rd International Conference on Machine Learning*, volume 306 of *Proceedings of Machine Learning Research*. PMLR, 2026. URL https://openreview.net/forum?id=kYqJ5gGjHE.

Lingyun Yang, Yuxiao Wang, Shenghao Liang, Linfeng Yang, Daocheng Ying, Chunbo You, Rui Zhang, Luping Wang, Yinghao Yu, Guodong Yang, and Liping Zhang. Are LLM-generated GPU kernels production-ready? a trace-driven benchmark and optimization agent. arXiv preprint arXiv:2607.14541, 2026. URL https://arxiv.org/abs/2607.14541.

Z.ai. GLM-4.7-FP8. Hugging Face model card, 2025. URL https://huggingface.co/zai-org/GLM-4.7-FP8. 358B-parameter FP8 checkpoint.

Genghan Zhang, Shaowei Zhu, Anjiang Wei, Zhenyu Song, Allen Nie, Zhen Jia, Nandita Vijaykumar, Yida Wang, and Kunle Olukotun. AccelOpt: A self-improving LLM agentic system for AI accelerator kernel optimization. In *Proceedings of Machine Learning and Systems*, volume 8, pp. 541–568. MLSys, 2026.

Zijian Zhang, Rong Wang, Shiyang Li, Yuebo Luo, Mingyi Hong, and Caiwen Ding. CudaForge: An agent framework with hardware feedback for CUDA kernel optimization. arXiv preprint arXiv:2511.01884, 2025. URL https://arxiv.org/abs/2511.01884.

Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 863–879. USENIX Association, 2020. URL https://www.usenix.org/conference/osdi20/presentation/zheng.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems*, volume 37, pp. 62557–62583, 2024. doi: 10.52202/079017-2000. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/724be4472168f31ba1c9ac630f15dec8-Abstract-Conference.html.

# A Optimization Algorithms

## A.1 GraphAgent Algorithms

Algorithm 1 describes the whole-model optimization driver across runtime graph groups. Algorithm 2 shows the staged optimization process within each graph group, while Algorithm 3 presents the shared backend for optimizing, validating, and integrating selected regions.

## A.2 KernelAgent Algorithms

The algorithms below summarize the optimization procedure of KernelAgent. Algorithm 4 presents the overall iterative search process, including candidate selection, memory update, profiling, and pool maintenance across rounds. Algorithm 5 details a single optimization round, where multiple directions are explored from each parent candidate, individual optimization units are evaluated independently, and successful units are further combined.

**Algorithm 1** GRAPHAGENT whole-model optimization procedure

**Require:** Model $M$, serving configuration $C$, hardware profile $H$, fusion-excluded nodes $E$, operator library $L$, round budgets $\mathbf{R} = (R_1, R_2, R_3)$
**Ensure:** Optimized serving runtime $\mathcal{O}^*$, end-to-end verification result $V_{\text{e2e}}$, and measured runtime $T_{\text{e2e}}$

$\mathcal{T} \leftarrow$ LAUNCHANDCAPTURERUNTIME$(M, C)$ ▷ launch SGLang and capture layer runtime graphs
$\mathcal{G} \leftarrow$ GROUPBYGRAPHHASH$(\mathcal{T})$
$\mathcal{G} \leftarrow$ SORTBYLAYERCOUNT$(\mathcal{G})$ ▷ groups covering more layers are optimized first
$\mathcal{K} \leftarrow \emptyset$ ▷ reusable optimizations from previously optimized hash groups
$\mathcal{O} \leftarrow \emptyset$ ▷ optimized graph for each hash group
**for** $i = 1$ **to** $|\mathcal{G}|$ **do**
  $g \leftarrow \mathcal{G}[i]$
  $G_g \leftarrow$ REPRESENTATIVEGRAPH$(g)$ ▷ same graph structure and tensor shapes within each hash group
  $\mathcal{H}_g \leftarrow \emptyset$ ▷ optimization history for the current hash group
  **if** $i > 1$ **then**
    $\mathcal{X}_g \leftarrow$ MATCHPREVIOUSOPTIMIZATIONS$(G_g, \mathcal{K})$ ▷ Phase 0: cross-group reuse
    $\mathcal{A}_0 \leftarrow \emptyset$
    **for all** matched optimization $m \in \mathcal{X}_g$ **do**
      $x_m \leftarrow$ BUILDREUSEINTERFACE$(G_g, m)$ ▷ adapt the matched region to the current graph
      $t_m^{\text{base}} \leftarrow$ BENCHMARKCURRENTREGION$(G_g, x_m)$
      $y_m \leftarrow$ ATTACHREUSABLEIMPLEMENTATION$(x_m, m)$
      $t_m \leftarrow \perp$
      $accepted_m \leftarrow$ **false**
      $(ok_m, y_m) \leftarrow$ VERIFYCORRECTNESS$(y_m, x_m)$
      **if** $ok_m$ **then**
        $t_m \leftarrow$ BENCHMARK$(y_m)$
        **if** $t_m < t_m^{\text{base}}$ **then**
          $\mathcal{A}_0 \leftarrow \mathcal{A}_0 \cup \{y_m\}$
          $accepted_m \leftarrow$ **true**
        **end if**
      **end if**
      $\mathcal{H}_g \leftarrow$ RECORDOUTCOME$(\mathcal{H}_g, 0, x_m, y_m, t_m^{\text{base}}, t_m, ok_m, accepted_m)$
    **end for**
    $G_g \leftarrow$ INTEGRATEACCEPTED$(G_g, \mathcal{A}_0)$
  **end if**
  $B_g \leftarrow$ BENCHMARKGRAPH$(G_g)$ ▷ whole-layer and per-node runtime measurements
  $(G_g, B_g, \mathcal{H}_g) \leftarrow$ OPTIMIZEGRAPHGROUP$(G_g, B_g, H, E, L, \mathbf{R}, \mathcal{H}_g)$
  $\mathcal{K} \leftarrow$ UPDATEREUSABLEOPTIMIZATIONS$(\mathcal{K}, g, G_g, \mathcal{H}_g)$
  $\mathcal{O} \leftarrow \mathcal{O} \cup \{(\text{HASH}(g), G_g)\}$
**end for**
$(\mathcal{O}^*, V_{\text{e2e}}, T_{\text{e2e}}) \leftarrow$ PUTBACKANDEVALUATE$(M, C, \mathcal{O})$ ▷ relaunch SGLang with optimized runtime graphs
**return** $(\mathcal{O}^*, V_{\text{e2e}}, T_{\text{e2e}})$

**Algorithm 2** GRAPHAGENT staged optimization for one runtime graph group

**function** OPTIMIZEGRAPHGROUP($G, B, H, E, L, \mathbf{R}, \mathcal{H}$)
  **for** $\phi = 1$ **to** $3$ **do**
    **for** $r = 1$ **to** $R_\phi$ **do**
      **if** $\phi = 1$ **then**
        $(\mathcal{P}, stageDone) \leftarrow$ PARTITIONGUIDEDSELECTION$(G, B, H, E, \mathcal{H})$ ▷ partition the current graph; inspect overlaps and roll back conflicting proposals
      **else if** $\phi = 2$ **then**
        $(\mathcal{P}, stageDone) \leftarrow$ EXPLORATIONGUIDEDSELECTION$(G, B, H, E, \mathcal{H})$ ▷ explore new combinations; inspect overlaps and roll back conflicting proposals
      **else**
        $(\mathcal{P}, stageDone) \leftarrow$ LIBRARYGUIDEDSELECTION$(G, L, H, \mathcal{H})$ ▷ match non-overlapping regions to expert-optimized operators
      **end if**
      **if** $|\mathcal{P}| = 0$ **then**
        **break**
      **end if**
      $(G, B, \mathcal{H}) \leftarrow$ OPTIMIZESELECTEDREGIONS$(G, \mathcal{P}, \phi, \mathcal{H})$
      **if** $stageDone$ **then**
        **break**
      **end if**
    **end for**
  **end for**
  **return** $(G, B, \mathcal{H})$
**end function**

**Algorithm 3** Shared backend for selected graph regions

**function** OPTIMIZESELECTEDREGIONS($G, \mathcal{P}, \phi, \mathcal{H}$)
  $G_{\text{cur}} \leftarrow G$ ▷ optimized graph at the beginning of this round
  $\mathcal{X} \leftarrow$ BUILDINTERFACESANDVERIFIERS($\mathcal{P}, G_{\text{cur}}$) ▷ region semantics, correctness checks, and proposal metadata
  **for all** region interface $x \in \mathcal{X}$ **do**
    $t_x^{\text{base}} \leftarrow$ BENCHMARKCURRENTREGION($G_{\text{cur}}, x$)
  **end for**
  **if** $\phi < 3$ **then**
    $\mathcal{I} \leftarrow \emptyset$ ▷ optimization inputs for KERNELAGENT
    **for all** region interface $x \in \mathcal{X}$ **do**
      **if** CONTAINSINTEGRATEDREGION($x, G_{\text{cur}}$) **then**
        $z_x \leftarrow$ COMPOSEINTEGRATEDCODE($x, G_{\text{cur}}$)
        $input_x \leftarrow$ COMPLETEINPUTWITHLLM($x, z_x$)
      **else**
        $input_x \leftarrow$ GENERATEINPUTWITHLLM($x, G_{\text{cur}}$)
      **end if**
      $\mathcal{I} \leftarrow \mathcal{I} \cup \{(x, input_x)\}$
    **end for**
    $\mathcal{Y} \leftarrow$ OPTIMIZEBATCHWITHKERNELAGENT($\mathcal{I}$) ▷ one best implementation is returned for each region
  **else**
    $\mathcal{Y} \leftarrow$ ADAPTANDATTACHLIBRARYOPERATORS($\mathcal{X}$) ▷ use the library match retained in each region proposal
    **for all** library result $y \in \mathcal{Y}$ **do**
      $x \leftarrow$ SOURCEINTERFACE($y$)
      $y \leftarrow$ RESOLVELIBRARYDEFINITIONCONFLICTS($y, x, G_{\text{cur}}$)
    **end for**
  **end if**
  $\mathcal{A} \leftarrow \emptyset$ ▷ accepted optimized regions
  **for all** result $y \in \mathcal{Y}$ **do**
    $x \leftarrow$ SOURCEINTERFACE($y$)
    $t_y \leftarrow \perp$
    $accepted_y \leftarrow$ **false**
    $(ok_y, y) \leftarrow$ VERIFYCORRECTNESS($y, x$)
    **if** $ok_y$ **then**
      $t_y \leftarrow$ BENCHMARK($y$)
      **if** $t_y < t_x^{\text{base}}$ **then**
        $\mathcal{A} \leftarrow \mathcal{A} \cup \{y\}$
        $accepted_y \leftarrow$ **true**
      **end if**
    **end if**
    $\mathcal{H} \leftarrow$ RECORDOUTCOME($\mathcal{H}, \phi, x, y, t_x^{\text{base}}, t_y, ok_y, accepted_y$)
  **end for**
  $G \leftarrow$ INTEGRATEACCEPTED($G_{\text{cur}}, \mathcal{A}$) ▷ selected regions are pairwise non-overlapping within the round
  $B \leftarrow$ BENCHMARKGRAPH($G$) ▷ refresh whole-layer and per-node measurements
  **return** $(G, B, \mathcal{H})$
**end function**

**Algorithm 4** KERNELAGENT optimization driver

**Require:** Initial kernel candidate $c_0$, verification oracle $V$, maximum rounds $R$, beam width $B = 3$, directions per parent $D = 3$, pool size $K = 20$, candidate-retention multiplier $\tau = 2$, significant-improvement threshold $\epsilon$

**Ensure:** Best optimized kernel candidate $c^*$

$c_0 \leftarrow \text{EVALUATEANDPROFILE}(c_0, V)$
$\mathcal{C} \leftarrow \{c_0\}$ ▷ bounded candidate pool
$\mathcal{M} \leftarrow \emptyset$ ▷ dynamic memory: attempts, reflections, notes, reusable snippets
**for** $r = 1$ **to** $R$ **do**
  $\mathcal{P} \leftarrow \text{RANKCANDIDATES}(\mathcal{C}, \mathcal{M})[1 : \min(B, |\mathcal{C}|)]$
  $(\mathcal{N}, \mathcal{H}) \leftarrow \text{GENERATEROUND}(\mathcal{P}, \mathcal{M}, D, V, \tau, \epsilon)$
  $\mathcal{N} \leftarrow \{\text{PROFILE}(q) \mid q \in \text{DEDUPLICATE}(\mathcal{N})\}$ ▷ deep profiles guide subsequent rounds
  $\mathcal{M} \leftarrow \text{UPDATEMEMORY}(\mathcal{M}, \mathcal{H})$
  $\mathcal{C} \leftarrow \text{RETAINCANDIDATEPOOL}(\text{DEDUPLICATE}(\mathcal{C} \cup \mathcal{N}), \mathcal{M}, K)$ ▷ runtime and exploration evidence
  $c^* \leftarrow \arg\min_{c \in \mathcal{C}} t_c$
  **if** $\text{REACHROOFLINE}(c^*)$ **then**
    **break**
  **end if**
**end for**
**return** $\arg\min_{c \in \mathcal{C}} t_c$

**Algorithm 5** Round-level direction exploration, unit evaluation, and fusion

**function** GENERATEROUND($\mathcal{P}, \mathcal{M}, D, V, \tau, \epsilon$)
  **for all** parent candidate $p \in \mathcal{P}$ **in parallel do**
    $A_p \leftarrow$ ANALYZE($p$)
    $\mathcal{D}_p \leftarrow$ PROPOSEDIRECTIONS($p, A_p, \mathcal{M}, D$) ▷ target metric and suggested mechanism
    **for all** optimization direction $d \in \mathcal{D}_p$ **in parallel do**
      $\mathcal{U}_{p,d} \leftarrow$ GENERATEUNITPLANS($p, d, \mathcal{M}$)
      $\mathcal{N}_{p,d} \leftarrow \emptyset$ ▷ retained candidates
      $\mathcal{H}_{p,d} \leftarrow \emptyset$ ▷ attributed optimization evidence
      $\mathcal{S}_{p,d} \leftarrow \emptyset$ ▷ significantly improved units
      **for all** unit plan $u \in \mathcal{U}_{p,d}$ **in parallel do**
        $(ok_u, u) \leftarrow$ VALIDATEANDREVISEUNITPLAN($p, d, u$) ▷ bounded revision
        **if** $\neg ok_u$ **then**
          $\rho_u \leftarrow$ REFLECTINVALIDPLAN($p, d, u$)
          $\mathcal{H}_{p,d} \leftarrow \mathcal{H}_{p,d} \cup \{\rho_u\}$
          **continue**
        **end if**
        $q \leftarrow$ IMPLEMENTUNIT($p, u$) ▷ each unit starts from the same parent
        $(ok_q, q) \leftarrow$ VERIFYANDREPAIR($q, V$)
        **if** $ok_q$ **then**
          $t_q \leftarrow$ BENCHMARK($q$)
          $s_q \leftarrow$ CLASSIFYSIGNAL($t_q, t_p, \epsilon$) ▷ significantly improved, neutral, or regressed
          $\rho_q \leftarrow$ REFLECT($p, d, u, q, t_q, s_q$)
          $\mathcal{H}_{p,d} \leftarrow \mathcal{H}_{p,d} \cup \{\rho_q\}$
          **if** $t_q < \tau t_p$ **then**
            $\mathcal{N}_{p,d} \leftarrow \mathcal{N}_{p,d} \cup \{q\}$
          **end if**
          **if** $s_q =$ SIGNIFICANTLYIMPROVED **then**
            $\mathcal{S}_{p,d} \leftarrow \mathcal{S}_{p,d} \cup \{q\}$
          **end if**
        **else**
          $\rho_q \leftarrow$ REFLECTFAILURE($p, d, u, q$)
          $\mathcal{H}_{p,d} \leftarrow \mathcal{H}_{p,d} \cup \{\rho_q\}$
        **end if**
      **end for**
      **if** $|\mathcal{S}_{p,d}| \geq 2$ **then**
        $f \leftarrow$ FUSEUNITS($p, \mathcal{S}_{p,d}$)
        $(ok_f, f) \leftarrow$ VERIFYANDREPAIR($f, V$)
        **if** $ok_f$ **then**
          $t_f \leftarrow$ BENCHMARK($f$)
          $\rho_f \leftarrow$ REFLECTFUSION($p, d, \mathcal{S}_{p,d}, f, t_f$)
          $\mathcal{H}_{p,d} \leftarrow \mathcal{H}_{p,d} \cup \{\rho_f\}$
          **if** $t_f < \min_{q \in \mathcal{S}_{p,d}} t_q$ **then**
            $\mathcal{N}_{p,d} \leftarrow \mathcal{N}_{p,d} \cup \{f\}$
          **end if**
        **else**
          $\rho_f \leftarrow$ REFLECTFUSIONFAILURE($p, d, \mathcal{S}_{p,d}, f$)
          $\mathcal{H}_{p,d} \leftarrow \mathcal{H}_{p,d} \cup \{\rho_f\}$
        **end if**
      **end if**
    **end for**
    $\mathcal{N}_p \leftarrow \bigcup_{d \in \mathcal{D}_p} \mathcal{N}_{p,d}$
    $\mathcal{H}_p \leftarrow \bigcup_{d \in \mathcal{D}_p} \mathcal{H}_{p,d}$
  **end for**
  $\mathcal{N} \leftarrow \bigcup_{p \in \mathcal{P}} \mathcal{N}_p$
  $\mathcal{H} \leftarrow \bigcup_{p \in \mathcal{P}} \mathcal{H}_p$
  **return** $(\mathcal{N}, \mathcal{H})$
**end function**

# B Optimization Trace

## B.1 End-to-End Optimization Trace

To illustrate how the graph- and kernel-level components interact in practice, we trace the complete optimization of one representative layer. Starting from the original execution graph, each round operates on the graph produced by the preceding round. GraphAgent identifies candidate regions under the current graph structure, and KernelAgent searches for executable implementations for the selected regions. Candidate implementations are verified against the original computation and benchmarked on the target hardware. Only verified implementations that are accepted by the optimization procedure are folded back into the graph; rejected candidates leave the current graph unchanged. The rewritten graph, together with the accumulated optimization history, is then used as the input to the next round.

Figure B.1 summarizes this process. The initial execution contains 41 kernels with a measured layer latency of 198.303 $\mu$s. The first round introduces several local fusions across the pre-attention, output-projection, MLP, router, and MoE regions, reducing the executed graph to 25 kernels and 127.935 $\mu$s. The second round optimizes the already rewritten graph rather than restarting from the original graph. It discovers additional fusion boundaries and alternative implementations, producing a 20-kernel graph with a latency of 109.264 $\mu$s. The final round further coalesces the pre-attention region into a single fused implementation and refines several existing implementations, resulting in 18 kernels and 108.528 $\mu$s.

Not every part of the layer is rewritten. In particular, the attention kernels remain unchanged throughout the trace, while the surrounding projection, normalization, routing, and MoE computation are progressively transformed. This behavior is important because the optimization unit is the current execution graph rather than a predetermined list of fusion patterns: previously optimized regions can participate in later regions, whereas regions for which no profitable verified rewrite is found within the current implementation capabilities of KernelAgent remain intact.

Table B.1 provides a more detailed view of the principal structural changes. We report transformations at the region level rather than enumerating every kernel in the execution trace. A row therefore describes a change in the executed graph boundary or a corresponding implementation replacement; kernels not listed in a round either retain their previous implementation or are unaffected by that rewrite.

**What the trace demonstrates.** The trace highlights two properties of the optimization procedure. First, fusion decisions are *state dependent*: Round 2 does not optimize the same graph seen by Round 1, and Round 3 can form a larger region from operations and implementations produced by earlier rounds. For example, the pre-attention path evolves from several independent kernels, to multiple local fusions, to a reorganized GEMM/epilogue decomposition, and finally to `_fused_all_kernel`. Second, optimization is selective rather than requiring the entire layer to be rewritten. The attention kernels remain stable while surrounding regions continue to change. Together, these observations show how FusionAgent carries optimization decisions through successive graph states until no further rewrite is accepted under the current search stage.

## B.2 Selected Operator Regions

Table 3 evaluates four representative operator regions extracted from the optimization traces of the target models. *MoE gate + top-k + align* is extracted from Layer 3 of GLM-4.7-358B, while the other three regions are extracted from Layer 3 of LLaDA 2.2 Flash. These regions cover both relatively compact fusion opportunities and longer heterogeneous operator chains, including routing, alignment, quantization, GEMM, normalization, and positional encoding. We describe their computation semantics and workload configurations below.

**MoE alignment.** This region implements the alignment step used to reorganize token–expert assignments before grouped expert computation. Given top-$k$ expert indices, it flattens the token–expert assignments, groups them by expert, pads each expert segment to the required block size, and produces the aligned token indices, expert indices for each block, and the total padded token count.

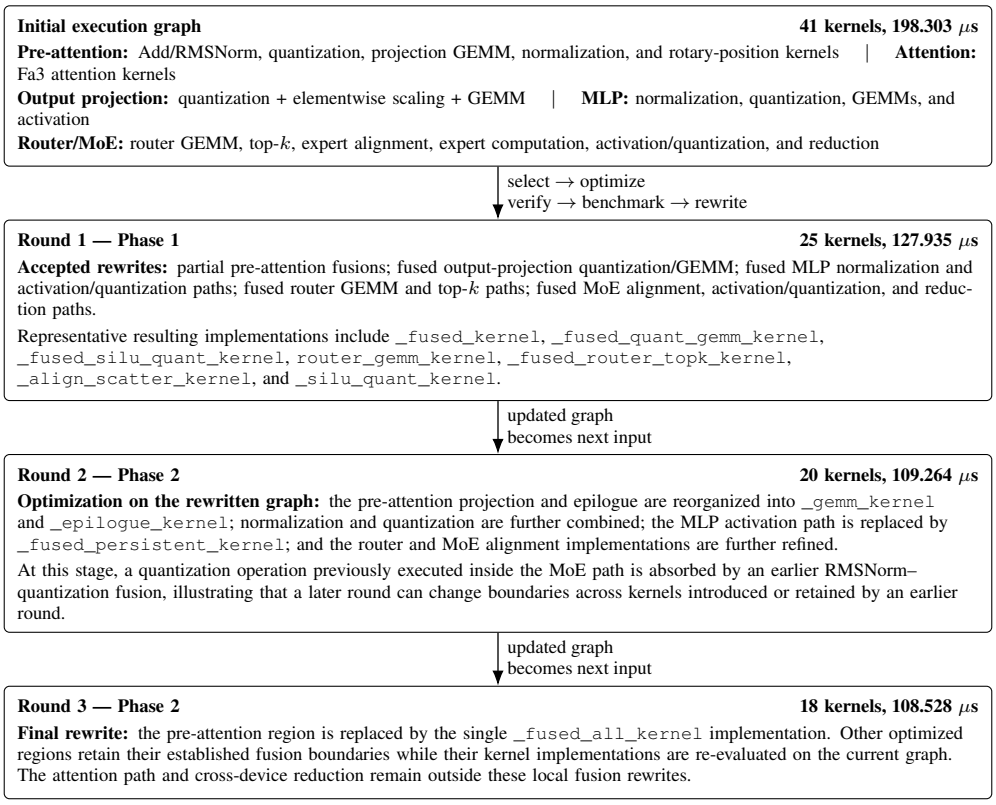



Figure B.1: End-to-end optimization trace for a representative layer. Each round starts from the graph produced by the previous round: GRAPHAGENT selects regions, KERNELAGENT searches for implementations, and verified performance-improving candidates are folded back into the graph. The figure reports the executed-kernel count and measured layer latency after each round. Only accepted graph changes are shown; unsuccessful candidates do not modify the graph. Since the current implementation of KERNELAGENT does not optimize multi-GPU communication operators, the reported latencies exclude `all_reduce` execution time.

The evaluated workload contains 32 tokens with top-$k = 8$, yielding 256 token–expert assignments over 257 experts with an alignment block size of 8.

**Quant + FP8 O-proj GEMM.** This region fuses activation quantization with the attention output-projection GEMM. A BF16 activation of shape $32 \times 512$ is first quantized to FP8 E4M3 using per-token group quantization with group size 128. The quantized activation is then multiplied by an FP8 output-projection weight of shape $4096 \times 512$ using block-wise activation and weight scales, and the final result is returned in BF16 with shape $32 \times 4096$. The resulting GEMM configuration is $M = 32$, $K = 512$, and $N = 4096$.

**MoE gate + top-k + align.** Extracted from Layer 3 of GLM-4.7-358B, this region covers the MoE routing pipeline from gate projection through expert selection and token alignment. It projects 32 token activations of hidden size 5120 to 160 expert logits, applies the routing activation and expert bias, performs grouped top-$k$ selection, and produces normalized routing weights for the selected experts. It then performs MoE block alignment to generate the expert-grouped token layout required by the downstream grouped GEMM. Each token selects nine experts, resulting in $32 \times 9 = 288$ token–expert assignments, with an alignment block size of 16.

**Add + RMSNorm + QKV GEMM + QK norm + RoPE.** Extracted from Layer 3 of LLaDA 2.2 Flash, this region represents a longer pre-attention pipeline. It first performs residual addition and RMS normalization on activations of shape $32 \times 4096$. The normalized activation is quantized

Table B.1: Principal accepted rewrites in the optimization trace of Figure B.1. Latencies in parentheses are median kernel execution times in microseconds for the resulting implementation.

| Round | Region | Resulting graph change / implementation |
|---|---|---|
| 1 | Pre-attention | Partial fusions produce `_fused_add_rmsnorm_kernel_masked` (1.600), `_fused_kernel` (6.976), and a fused rotary-position tail `fused_kernel` (1.600). |
| 1 | Output proj. | Quantization, elementwise scaling, and projection GEMM are replaced by `_fused_quant_gemm_kernel` (3.648). |
| 1 | MLP | The normalization path is replaced by `_fused_add_rmsnorm_kernel` (1.569); the activation–quantization path becomes `_fused_silu_quant_kernel` (1.344). |
| 1 | Router | The projection path becomes `router_gemm_kernel` (5.152), while the routing/top-$k$ path becomes `_fused_router_topk_kernel` (4.640). |
| 1 | MoE | Expert alignment becomes `_align_scatter_kernel` (2.592); activation and quantization become `_silu_quant_kernel` (1.456); the reduction tail is replaced by `fused_kernel` (1.824). |
| 2 | Pre-attention | The rewritten projection path is further reorganized into `_gemm_kernel` (4.960), and the remaining epilogue operations are combined into `_epilogue_kernel` (1.744). |
| 2 | MLP | RMSNorm and quantization are combined as `fused_rmsnorm_quant_kernel` (1.888); the activation/quantization/GEMM path is replaced by `_fused_persistent_kernel` (2.464). |
| 2 | Router | Existing fusion boundaries are retained while their implementations are replaced by `_router_gemm_splitk_kernel` (4.448) and `_router_topk_kernel` (4.224). |
| 2 | MoE | The alignment implementation becomes `_align_blocks_kernel` (2.112). A per-token quantization operation is eliminated as a separate launch after being absorbed into an earlier RMSNorm–quantization fusion. |
| 3 | Pre-attention | The previously rewritten pre-attention region is coalesced into the single `_fused_all_kernel` implementation (7.584). |
| 3 | Remaining regions | Existing fusion boundaries are retained while implementations are re-evaluated on the current graph; the complete graph reaches 18 executed kernels and 108.528 $\mu$s. |

with per-token FP8 group quantization and passed through an FP8 QKV projection with an output width of 768. The projected tensor is split into Q, K, and V components with widths 512, 128, and 128, respectively. Q and K are subsequently normalized per head and transformed using NeoX-style rotary positional encoding, after which the rotated K and V tensors are written to the KV cache. The workload uses four query heads, one KV head, head dimension 128, FP8 group size 128, and rotary dimension 64.

**Common initialization across agents.** For each operator region, all agentic baselines and our KERNELAGENT start from the same initial implementation. The starting implementation, input tensors, operator semantics, and workload configuration are kept identical across methods, so the comparison isolates differences in their optimization procedures rather than differences in initialization. Each agent is therefore given the same runnable implementation as its optimization starting point and independently searches for improved implementations under its respective optimization workflow.

### B.3 CASE STUDIES OF ECHO COMPONENT ABLATIONS

**Case study: exploration-aware ranking on Quant + FP8 O-proj GEMM.** The Quant + FP8 O-proj GEMM region illustrates why ranking candidates solely by current latency can steer the search toward a locally attractive but ultimately inferior implementation family. As shown in Table 4, removing exploration-aware ranking increases latency from $3.53\,\mu$s to $5.76\,\mu$s.

The best implementation found by the full KERNELAGENT uses a single kernel without split-$K$, atomics, or a separate reduction path. It quantizes the activation by group, keeps the quantized activations, weights, scales, and partial accumulations in registers, statically unrolls the $K$-group computation, and applies the final scaling and accumulation before writing the output. This avoids synchronization and scheduling overheads that are particularly expensive for the small $M = 32$ workload.

With performance-only ranking, the search is instead drawn early toward split-$K$ implementations, which obtain an immediate speedup from increased parallelism. Subsequent candidates continue refining this family with atomic accumulation and a counter-based finalization path. Although these candidates are competitive in the short term, the additional global reduction and synchronization eventually limit their performance. A structurally different branch—single-kernel execution without split-$K$ and with transposed FP8 MMA—does appear later in the search and reaches latency

comparable to the split-$K$ branch, but only near the end of the optimization budget and without the register-caching and static-unrolling structure of the final implementation.

This case illustrates the role of exploration-aware ranking in ECHO: candidate selection should account not only for current measured performance, but also for whether an implementation preserves a distinct and potentially productive optimization direction. Retaining such candidates prevents early gains from concentrating the search around a single implementation family and gives later rounds the opportunity to compose additional optimizations, leading to a better final result.

**Case study: unit-level attribution on MoE routing and alignment.** The MoE gate + top-$k$ + alignment region illustrates the effect of unit-level attribution. The full KERNELAGENT reaches $9.02\,\mu$s with two kernels: a gate GEMM followed by a fused routing–alignment kernel. The latter combines top-$k$ routing with expert alignment using compact bitmask publication and lightweight population-count-based finalization, while avoiding repeated intermediate-buffer initialization.

When plan decomposition is removed, and thus optimization changes can no longer be isolated into attribution units, the final implementation reaches $15.70\,\mu$s with five stages: gate GEMM, routing, prefix processing, expert-metadata construction, and token scattering. During optimization, implementations closer to the optimum also appear: one candidate already fuses routing and alignment, while another incorporates part of the alignment construction into routing. However, these candidates simultaneously introduce expensive finalization, initialization, or scattering work and therefore perform poorly.

Without unit-level attribution, each such candidate is evaluated as a whole, so its measured regression cannot be assigned to the specific hypothesis or mechanism responsible for it. A slow fused candidate can therefore produce misleading negative evidence for routing–alignment fusion even when the regression actually arises from its finalization or alignment realization. Plan decomposition separates independently testable changes into attribution units, allowing KERNELAGENT to preserve a promising optimization hypothesis while identifying the ineffective implementation choice. The resulting evidence is further retained in the Technique Library for reuse. This finer-grained attribution explains the $74.1\%$ latency increase observed without plan decomposition in Table 4.

**Case study: Technique Library on MoE alignment.** The MoE alignment region illustrates the effect of the Technique Library. As shown in Table 4, removing the library increases latency from $2.67\,\mu$s to $11.52\,\mu$s, the largest latency increase among the four evaluated regions.

The best implementation found by the full KERNELAGENT is specialized for this small workload. It replaces the conventional count–scan–scatter pipeline with a fused design that combines token scattering, expert-block assignment, and padding in one launch, redundantly recomputes small histogram and prefix information, avoids global atomics on the main path, and uses workload-dependent execution paths. Together, these choices reduce kernel launches, synchronization, and global-memory intermediates.

Without the Technique Library, the search instead converges to a conventional multi-stage implementation with separate counting, prefix computation, expert assignment, and scattering, incurring additional launches, synchronization, and atomic updates.

Inspecting the dynamic knowledge accumulated along the search trajectory shows that many of the transformations used by the best implementation have been distilled into reusable techniques. Representative entries include *Replicated Tiny Scan instead of Sync Launch*, which records when redundant local prefix computation is cheaper than a separate synchronization launch; *Concatenated PID-range Fusion*, which combines independent output-producing stages into one launch; *Block-local Pairwise Rank with Global Offset*, which derives stable local positions while retaining a global base offset; and *Size-gated Dual Path*, which selects between recomputation and materialized auxiliary state according to workload size. The library also accumulates techniques for balancing block size against duplicated work and for folding padding or initialization into an existing launch.

These entries capture reusable optimization observations rather than complete implementations, allowing later search rounds to exploit successful transformations without restricting exploration to previously seen kernels. The final implementation also contains a workload-specific specialization that has not yet been distilled into an independent library technique: for 257 experts, it processes 256 experts as the main vectorized group and handles the remaining expert separately, avoiding the

overhead of extending all expert-wise operations to the next padded width. This case shows how the dynamically accumulated Technique Library preserves and reuses effective optimization strategies across rounds, providing reusable guidance for subsequent search and helping KERNELAGENT reach better-performing implementations within a fixed optimization budget.