LLM4LLM: Bridging Kernel Benchmarks and Real Deployment via Closed-Loop Agentic Optimization

Hui Zeng[1,2], Pengfei Yang[1,4,*], Yanxin Chen[1], Fusong Ju[2], Xinran Wei[2,3,*]

[1]Xidian University, [2]Zhongguancun Academy, [3]Zhongguancun Institute of Artificial Intelligence
[4]National Key Laboratory of Advanced Communication Networks, Shijiazhuang, Hebei, P. R. China
[*]**Corresponding authors:** pfyang@xidian.edu.cn, weixinran@zgci.ac.cn

Abstract

Large language models have become increasingly capable agents for low-level code and kernel optimization, but isolated kernel benchmarks provide only a proxy for the deployment behavior that matters in language-model inference. We identify a benchmark-to-deployment gap: candidate kernels that appear correct and fast in standalone harnesses can exhibit different performance, safety, or phase behavior after integration into a real inference workload. We introduce LLM4LLM, a deployment-aware closed-loop optimization framework that starts from a target inference script, extracts phase-aware optimization tasks, searches with an experience-guided episodic agent, and accepts patches through in-model validation. Across ten language-model inference workloads on A100 and H100 GPUs, LLM4LLM improves end-to-end latency for every evaluated model, achieving 3.91×/6.98× geometric-mean speedups on A100/H100; as supporting kernel-level evidence, it also attains up to 2.745× GeoMean speedup on KernelBench Level 2.

1 Introduction

Recent progress in LLM-driven code generation has made automated kernel and program optimization a credible direction for performance engineering. Code-specialized models support synthesis, infilling, and editing (Chen et al., 2021; Wang et al., 2021). Competitive-programming and execution-feedback systems extend this ability to search-guided generation (Li et al., 2022; Le et al., 2022). Iterative agents improve programs through feedback and debugging traces (Madaan et al., 2023; Chen et al., 2023). Recent kernel benchmarks further show that LLMs can generate low-level GPU programs (Ouyang et al., 2025). Most existing evaluations, however, are centered on isolated kernels or extracted program harnesses: a candidate is compiled, executed on synthetic inputs, checked against a reference implementation, and ranked by standalone latency. For language-model inference, this measures an intermediate signal. The deployment objective is the behavior of the patched model under the target workload, including phase semantics, cache state, dispatch overhead, memory residency, and end-to-end latency.

We study the mismatch between benchmark-level optimization and model-level deployment. Autoregressive inference exposes this mismatch clearly. Prefill and decode have different shapes, cache states, memory-access patterns, and latency sensitivity; a candidate optimized for one phase can transfer unevenly to the other. Integration can also change the execution environment through shape guards, dispatch logic, allocator state, and interactions with existing high-performance kernels. We refer to this prediction mismatch as the *benchmark-to-deployment gap*. The gap appears as speedup reversal, deployment-time runtime failure, and phase-specific behavior that is absent from isolated qualification.

LLM4LLM[1] addresses this gap by making deployment context part of the optimization loop. Starting from a user-provided inference script, the system profiles real workloads and extracts optimization tasks at deployable module boundaries. It constructs phase-aware tasks for prefill and decode, searches for candidates through experience-guided episodic optimization, and accepts patches only after in-model validation. The episodic agent compresses useful experience from earlier attempts into concise constraints, allowing search to reuse successful patterns while discarding stale context. The resulting loop ties profiling, generation, verification, acceptance, and deployment into a single optimization process.

Our contributions are:

[1]Code is available at https://github.com/hzeng2000/LLM4LLM.

- We characterize the benchmark-to-deployment gap for LLM-based kernel optimization through a taxonomy of failure modes, showing how isolated qualification can diverge from deployment-time acceptance.

- We introduce LLM4LLM, a closed-loop agentic optimization framework that combines real-model profiling, phase-aware task formulation, experience-guided episodic search, and in-model acceptance.

- We evaluate LLM4LLM on real inference workloads and KernelBench Level 2, showing end-to-end gains across transformer, state-space, and recurrent language-model families with 3.91×/6.98× geometric-mean speedups on A100/H100, and GeoMean kernel speedups of 2.745×/2.628× on KernelBench Level 2.

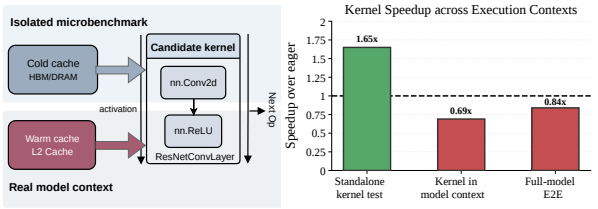


Figure 1: Divergent isolated and deployment outcomes for an optimized `ResNetConvLayer`. The left panel contrasts standalone cold-cache benchmarking with in-model warm-context execution for the same candidate kernel. The right panel reports speedup from standalone kernel testing, the same kernel measured inside the model context, and the resulting full-model end-to-end execution.

2 The Benchmark-to-Deployment Gap

2.1 Motivating Evidence

We begin with a representative case that isolates the effect of execution context, shown in Figure 1. A convolution layer extracted from a standard vision backbone is optimized by an LLM-based agent and evaluated in two settings. In the isolated setting, the candidate is compiled as a standalone module, invoked on freshly allocated random inputs, and timed independently. In the deployment setting, the same candidate is patched into the full model and evaluated in the target inference workload. The isolated benchmark reports a substantial speedup, while the integrated model becomes slower end to end.

The mechanism is memory residency. In the standalone benchmark, the operator pays the cost of loading cold inputs from memory. Inside the model, the same operator consumes activations produced immediately by the preceding layer, so the baseline benefits from warm-cache execution and producer-consumer locality. A memory-access pattern that improves cold standalone timing can lose its advantage once the surrounding model supplies different cache state and scheduling context. This case shows how proxy measurements can misrank candidates before deployment.

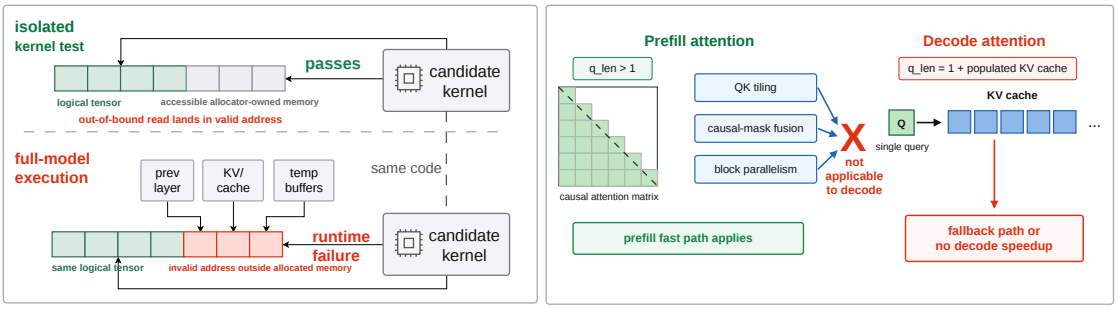
The same gap also appears in forms that are specific to deployability and autoregressive inference. First, a candidate may pass isolated correctness while carrying a latent memory-safety error. Standalone tensors are often allocated with adjacent mapped memory, so an out-of-bounds read can return a value without faulting. The allocator state of a full model can place the same access in a faulting region, producing a deployment-time CUDA error. Second, phase specialization can make a kernel-level gain local to only one part of generation. For example, a candidate that optimizes attention during prefill may be applicable when `q_len > 1`, while decode attention runs with `q_len = 1` and a populated KV cache. The prefill fast path can therefore fall back, or provide no benefit, in the decode phase.

These cases establish three deployment concerns: performance transfer, runtime validity, and phase-correct behavior. They also motivate a distinction between isolated qualification and deployment-time acceptance, as illustrated in Figure 2.

2.2 Formulation and Implications

We distinguish four evaluation stages. *Isolated qualification* checks numerical correctness and latency in a standalone harness. *Search-time validation* strengthens the proxy by adding real inputs, runtime-safety gates, or in-context timing. *Deployment-time acceptance* evaluates a candidate after it has been inserted into the target model instance. *Final end-to-end evaluation* measures the patched model under the full workload.

The benchmark-to-deployment gap is the mismatch between isolated qualification and deployment-time acceptance. Execution context, phase behavior, and integration constraints all contribute to this mismatch. Isolated benchmarks remain valuable because they allow rapid candidate generation and screening, while deployment-time



(a) Runtime invalidity

(b) Phase/cache-state mismatch

Figure 2: Deployment-specific manifestations of the benchmark-to-deployment gap. The left panel shows how a latent out-of-bounds access can pass isolated qualification under a fresh allocator state yet fail after integration into the full model runtime. The right panel shows that an attention candidate specialized for prefill can be inapplicable to decode attention, so a kernel-level gain does not necessarily transfer across generation phases.

acceptance determines whether a candidate improves the target model. This leads to a methodological requirement: candidate generation, validation, and patch acceptance form a closed loop around the real workload.

3 Method

Figure 3 summarizes the LLM4LLM optimization loop. Starting from a target inference script, the system profiles the real workload, extracts phase-aware optimization tasks, searches with an experience-guided agent, and accepts patches through model-integrated validation. We denote the target model instance by M, the deployment workload induced by the inference script by $\mathcal{W}$, and the set of autoregressive phases by $\mathcal{P}$. Here M is a concrete model instance, while $\mathcal{W}$ and $\mathcal{P}$ are structured collections. For a module instance m and phase $p \in \mathcal{P}$, profiling records the module time $t(m,p)$ and the total phase latency $T(p)$. LLM4LLM selects deployable optimization units using the workload-weighted hotspot score

$$s(m) = \sum_{p \in \mathcal{P}} \omega_p \frac{t(m,p)}{T(p)}, \tag{1}$$

where ω_p is determined by the measured phase frequency or by the evaluation workload. The score is used for task extraction, and final acceptance is made inside the target model.

We next detail the three stages of this loop: phase-aware task extraction, experience-guided search, and deployment-time acceptance.

3.1 Hotspot Discovery and Phase-Aware Extraction

LLM4LLM starts from the inference script supplied by the user. The script defines the workload, input regime, runtime path, and performance objective. The system profiles this execution hierarchically and selects semantic module instances whose replacement can affect end-to-end latency. A module is considered deployable when it has a stable call boundary, reproducible input and output tensors, and a fallback implementation for unoptimized regimes.

For each selected module, LLM4LLM serializes an optimization task containing representative inputs, output references, shape information, module-family metadata, hardware scope, and phase tags. For autoregressive inference, prefill and decode are represented as separate tasks when their execution semantics diverge. When both phases are optimized, extraction also records a dispatch template that later reassembles phase-specialized candidates into a single patched module. The task preserves the deployment facts that affect validity, including tensor layout, cache state, phase predicate, and observed shape regimes.

3.2 Experience-Guided Episodic Optimization

The optimizer searches through repeated generate-verify-decide episodes. Within an episode, the agent proposes candidate kernels, compiles them, runs correctness checks, measures latency, and inspects failures. Each episode therefore produces a concrete validation trace: candidate code, compiler diagnostics, numerical errors, runtime failures, and latency measurements. LLM4LLM uses this trace to decide whether the current search context is still productive or has converged to local repair of a narrow failure mode. For a candidate c on task τ, search-time validation assigns the utility

$$u(c,\tau) = \frac{t_{\text{ref}}(\tau)}{t_c(\tau)} \mathbf{1}\{\text{err}(c,\tau) \le \epsilon,\ \text{safe}(c,\tau)\}, \tag{2}$$

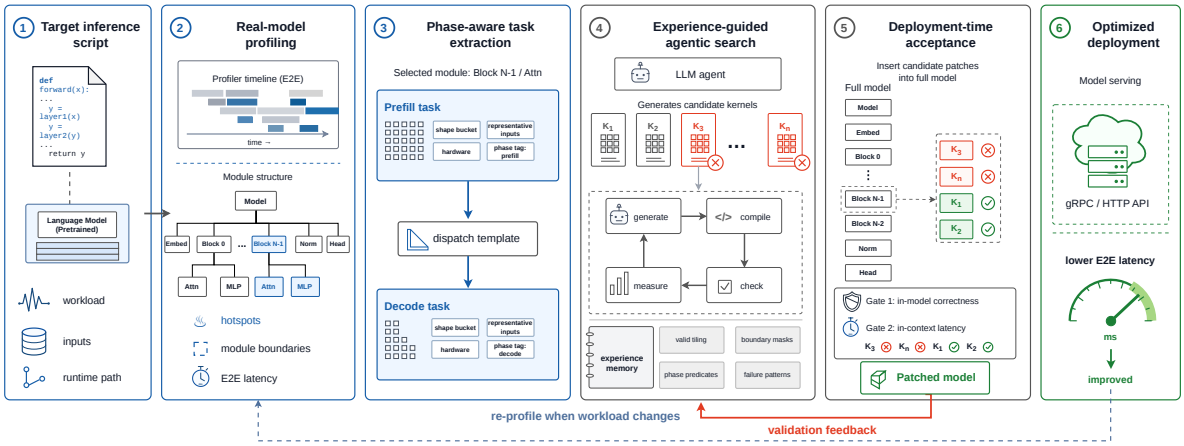


Figure 3: Overview of LLM4LLM. The framework starts from a target inference script, identifies deployable hotspots through real-model profiling, constructs phase-aware tasks, searches for candidate kernels with experience-guided agentic optimization, and accepts patches only after in-model correctness and latency validation. Validation feedback is returned to the search loop, while accepted patches produce the optimized deployment.

where t_{ref} is the reference task latency, t_c is the candidate latency, $\mathbf{1}\{\cdot\}$ is the indicator function, err is the numerical error against recorded outputs, and safe denotes compilation and runtime-safety checks. This utility ranks promising candidates within the extracted task; deployment-time acceptance remains the final decision.

At episode boundaries, the system distills the validation trace into compact experience. The distilled record contains durable constraints and search evidence, including valid tiling choices, boundary-mask requirements, phase predicates, numerical constraints, failure signatures, and the best observed performance regime. The next episode is initialized with the original task specification and this experience record; the full turn-by-turn transcript is archived. The summary for episode r is a bounded record E_r that stores accepted constraints, rejected failure modes, and the best candidate family observed so far. Restarting from the task plus E_r preserves useful evidence while reducing the influence of long local repair trajectories.

Figure 4 illustrates this mechanism on a representative optimization episode. This design treats validation feedback as reusable optimization evidence. Episode-level memory captures task-specific lessons, while family-scoped memory captures patterns that transfer across related modules, shape buckets, and hardware targets. The search process remains grounded in observed verification constraints while allowing later episodes to explore implementation structures beyond earlier local edits.

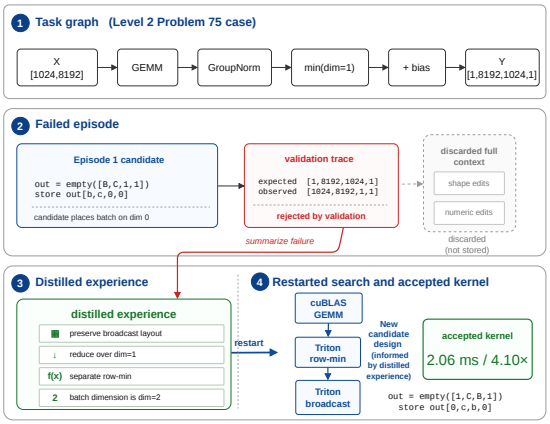


Figure 4: Experience-guided episodic optimization. Candidate generation and validation form a bounded episode. At the episode boundary, LLM4LLM summarizes the useful validation evidence into compact experience, archives the transient repair history, and restarts search from the task specification plus the distilled record.

3.3 Deployment-Time Acceptance and Patching

Search produces qualified candidates; deployment-time acceptance decides which candidates enter the model. LLM4LLM ranks candidates by module compatibility, shape coverage, phase compatibility, and predicted impact. Each candidate is inserted into the target model instance and evaluated for in-context correctness and latency. Accepted candidates become patches; rejected candidates provide feedback for subsequent search. Let $\ell(c) = L(M[c], \mathcal{W})/L(M, \mathcal{W})$ be normalized deployment latency, and let $\mathcal{C}_{\text{dep}} = \{c \in \mathcal{C} : \mathrm{ok}_{\text{dep}}(c)\}$ be the candidates passing in-model correctness, runtime compatibility, and shape-guard

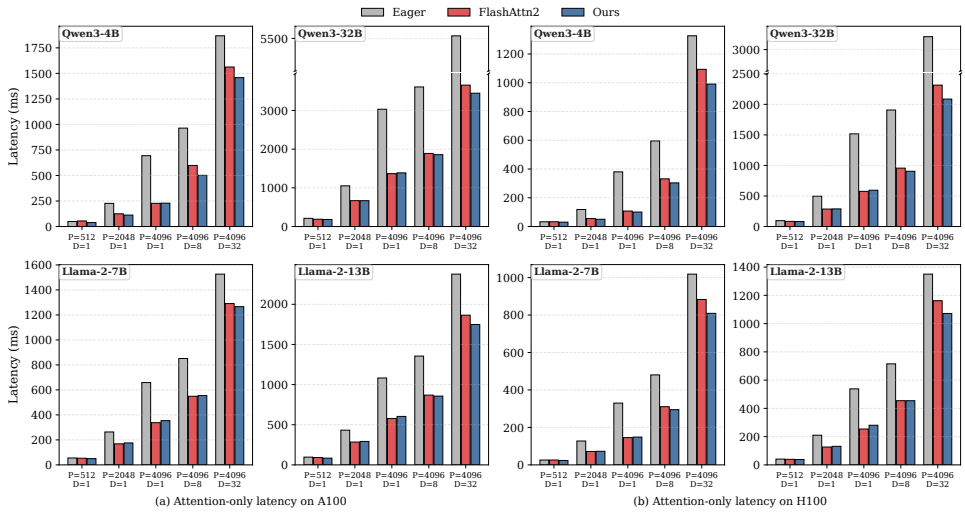


Figure 5: Scope-matched comparison on transformer-family attention workloads. Each panel reports latency for eager execution, LLM4LLM, and expert deployment kernels across representative workload settings on A100 and H100. Lower is better.

coverage. Deployment-time acceptance selects

$$c^\star = \arg\min_{c \in \mathcal{C}_{\text{dep}}} \ell(c), \qquad \ell(c^\star) \le 1 - \delta. \quad (3)$$

where δ is the minimum deployment improvement required to accept a patch. Equation 3 ties candidate generation to the evaluation objective.

For phase-specialized replacements, the patched module uses the dispatch template generated during extraction. Runtime guards preserve shape compatibility, and fallback paths preserve execution for unoptimized regimes. The final patch therefore reflects both the agent's generated implementation and the deployment constraints of the target workload.

4 Evaluation

4.1 Experimental Setup

We evaluate LLM4LLM on language-model inference workloads spanning transformer-family models, Mamba-family state-space models, and RecurrentGemma. Experiments are conducted on A100 and H100 GPUs. The primary metric is end-to-end latency of the target inference workload, reported together with speedup over eager execution. LLM4LLM starts optimization from the eager PyTorch execution path. For scope-matched comparisons, we evaluate attention-only and mixer-only settings against strong deployment baselines, including FlashAttention for attention and Mamba fast paths for state-space mixers. Here, Mamba fast paths refer to hand-optimized kernels used by the Mamba and Mamba-2 implementations: `mamba_ssm` for selective-scan and state-space mixer execution, and `causal-conv1d` for causal depthwise convolution (Gu and Dao, 2023; Dao and Gu, 2024; Dao, 2024). All reported patches pass model-integrated correctness checks before latency measurement. The evaluation follows the same acceptance path used by the method: candidates are validated on extracted tasks, inserted into the model instance, and measured through the target inference script. We report latency because the objective includes launch overhead, guards, cache behavior, and interactions with surrounding model code.

4.2 End-to-End Results on Language-Model Families

Table 1: End-to-end latency (ms) across language-model families on A100 and H100. Lower latency and higher speedup are better.

Model	A100			H100		
	Eager	Ours	Speedup	Eager	Ours	Speedup
Qwen3-4B	693.9	229.0	3.03×	380.2	100.7	3.78×
Qwen3-32B	3032.3	1387.1	2.19×	1516.7	594.4	2.55×
Llama-2-7B	658.8	353.9	1.86×	329.8	148.4	2.22×
Llama-2-13B	1082.2	603.7	1.79×	538.1	280.7	1.92×
Mamba (130M)	626.4	28.0	22.37×	537.7	16.4	32.79×
Mamba (2.8B)	959.2	118.5	8.09×	1446.4	50.5	28.64×
Mamba2 (130M)	198.5	33.0	6.02×	137.8	22.0	6.26×
Mamba2 (2.7B)	665.4	43.7	15.23×	445.0	31.7	14.04×
RecurrentGemma-2B	329.4	180.2	1.83×	287.9	26.8	10.74×
RecurrentGemma-9B	712.5	575.5	1.24×	409.8	54.3	7.55×

Table 1 shows that the closed-loop optimization produces end-to-end gains across all evaluated families. These numbers are measured after patch insertion, so they reflect the realized effect of candidate kernels together with dispatch overhead, shape guards, cache state, and surrounding model code. The transformer-family results show steady improvements on both GPUs. For these models, pro-

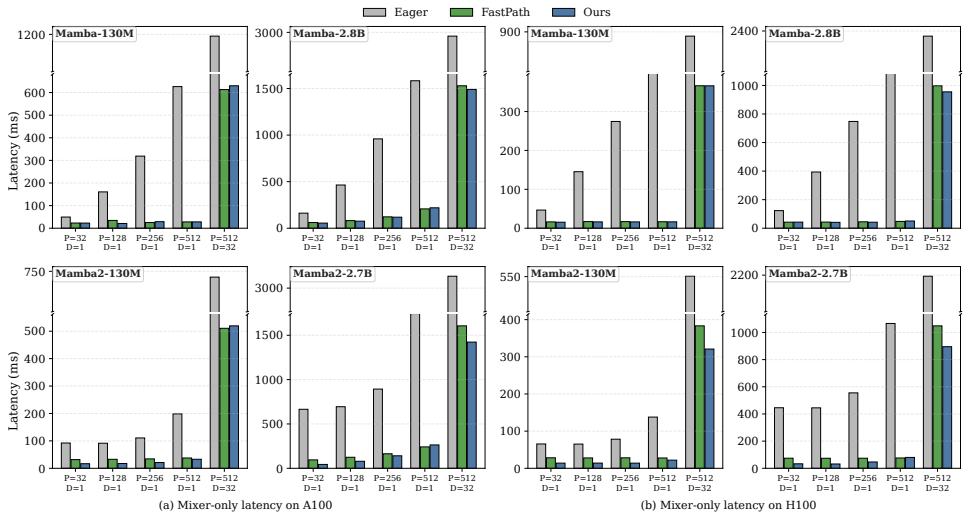


Figure 6: Scope-matched comparison on state-space mixer workloads. Each panel reports latency for eager execution, LLM4LLM, and Mamba-family fast paths based on `mamba_ssm` and `causal-conv1d` across representative workload settings on A100 and H100. Lower is better.

filing usually selects attention-dominated regions and adjacent tensor operations whose cost remains visible after existing fused attention paths are enabled. The gains therefore reflect deployment-level replacement boundaries and model-integrated acceptance.

The state-space and recurrent families show a different pattern. Mamba-family workloads concentrate latency in mixer modules whose computation is regular enough for generated Triton kernels to cover a large share of the inference path. RecurrentGemma exposes another profile: the dominant recurring module is `RecurrentGemmaRglru`, whose cost is tied to gated recurrent updates, state movement, and phase-dependent memory behavior. This observation motivates the first profiling stage of LLM4LLM. The dominant deployable kernel is model-family dependent: transformer traces emphasize attention regions, Mamba traces emphasize state-space mixers, and RecurrentGemma traces emphasize the RGLRU recurrent core. A fixed attention-first policy allocates budget poorly for these families; profiling-first extraction directs search to the modules that dominate the actual workload. The A100–H100 differences further show that the accepted patch depends on both model structure and hardware execution context.

4.3 Comparisons with Strong Deployment Baselines

Figures 5 and 6 compare LLM4LLM against strong deployment baselines under matched optimization scope. The attention comparison evaluates the setting where expert kernels are especially mature. LLM4LLM reaches competitive latency in several prompt/decode regimes, while the remaining gaps identify workload shapes where specialized attention implementations retain an advantage. This result is useful for deployment because it separates two questions: whether an LLM-generated candidate can execute correctly inside the model, and whether the measured attention region is the best use of the search budget for that model.

The mixer comparison gives complementary evidence on a family with a different dominant operation. When profiling selects state-space mixer boundaries, generated replacements can absorb surrounding reshapes, projections, and elementwise updates that are outside the scope of a single vendor or library kernel. This wider deployable boundary explains why LLM4LLM can approach or improve over the hand-optimized `mamba_ssm` and `causal-conv1d` paths in several regimes. Together with the RecurrentGemma results in Table 1, the scope-matched figures support a profiling-driven view of optimization: attention, state-space mixers, and recurrent RGLRU kernels each become the right target only when they dominate the measured workload. The deployment baselines therefore serve as strong references for their own scopes, while the closed-loop acceptance step determines which generated patch improves the full inference path.

4.4 KernelBench Comparison

KernelBench provides supporting evidence for kernel-level capability and agent search behavior while the preceding experiments measure deploy-

Table 2: KernelBench Level-2 comparison on A100 and H100 GPUs. Pass, Fast1, and Fast2 are reported as percentages. Mean and GeoMean are speedups over PyTorch eager unless otherwise specified.

A100-SXM4-80GB							H100-80GB-HBM3						
Method	**Lang.**	**Pass**	**Fast1**	**Fast2**	**Mean**	**GeoMean**	**Method**	**Lang.**	**Pass**	**Fast1**	**Fast2**	**Mean**	**GeoMean**
CUDA-L1	CUDA	100	98	–	3.55	–	CUDA-L1	CUDA	96	96	–	6.64	–
KernelSkill	CUDA	100	100	–	2.82	–	KernelBlaster	CUDA	81	84.85	–	10.223	2.592
STARK (A100-40G)	CUDA	100	100	–	2.69	–	QiMeng-Kernel	Triton	99	86	12	1.28	–
QiMeng-Kernel	Triton	99	66	8	1.22	–	AI CUDA Engineer	CUDA	82	61.33	–	1.356	1.214
LLM4LLM	Triton	100	97	39	22.157	2.745	LLM4LLM	Triton	99	95	41	15.847	2.628

ment transfer.

We first compare LLM4LLM with recent LLM-based kernel optimization systems on KernelBench Level 2. The comparison includes CUDA-L1 and KernelSkill (Li et al., 2025; Sun et al., 2026). It also covers STARK and QiMeng-Kernel (Dong et al., 2025; Zhu et al., 2025), as well as KernelBlaster and AI CUDA Engineer (Dong et al., 2026; Lange et al., 2025). Unless otherwise noted, rows are evaluated on the full KernelBench Level-2 split. Because existing papers report results with different aggregation conventions, Table 2 keeps arithmetic and geometric means as separate columns while standardizing the most common correctness and fast-p metrics. The comparison separates correctness, broad speedup, and heavy-tailed arithmetic gains, which are often conflated in aggregate kernel benchmark reports. LLM4LLM delegates exploit detection in isolated KernelBench runs to the benchmark harness: unconstrained LLMs can produce benchmark-exploiting candidates that hard-code fixed-test outputs or bypass the intended computation. This is one reason our primary evidence comes from deployment evaluations, where candidates are patched into real model code and accepted through model-integrated correctness and end-to-end latency.

4.5 Ablation Study

We further isolate the contribution of sampling, iterative refinement, and LLM4LLM's restart strategy across GPT-5.4, Claude Sonnet 4.6, and GLM-5. Each row reports one complete evaluation over the 100 KernelBench Level-2 tasks. For each task, we apply the candidate budget shown in the method name, retain the fastest correct candidate after KernelBench warmup and repeated timing, and aggregate the resulting task-level speedups using Mean, GeoMean, P50, and P75. Table 3 reports the same method grid against PyTorch eager and `torch.compile`. The first three rows in each model block follow KernelBench-style sampling and execution-feedback iteration (Ouyang et al., 2025), separating candidate diversity from refinement. Sample-10 improves substantially over Sample-1, indicating that independent diversity is a strong baseline for finding compilable and occasionally fast kernels. Iter-10 reaches a 100% pass rate for all three models, while its GeoMean and Fast2 metrics trail the stronger sampling runs in several settings; the search often spends many turns repairing one trajectory after the first viable implementation.

The LLM4LLM variants add deployment-aware task construction, validation feedback, and episodic restart with compact experience. With restart, the 15-trial setting gives the best GeoMean for all three models against eager execution (2.153, 2.546, and 2.745) and also improves the `torch.compile` comparison. The percentile columns contextualize the heavy-tailed arithmetic means: GLM-5 reaches P50/P75 speedups of 1.680/5.539 against eager and 1.434/5.281 against `torch.compile`, while Claude Sonnet 4.6 reaches 1.683/5.017 against eager. Comparing "w/o restart" with full LLM4LLM across all three models shows that restart contributes beyond a larger optimization budget, supporting the episodic design in Section 3.2. Increasing the budget from 10 to 15 trials yields consistent gains in GeoMean and Fast2, so the 15-trial configuration is used as the strongest KernelBench setting.

5 Related Work

5.1 Tensor Program and Kernel Optimization

Compiler and scheduling systems generate efficient tensor programs across operator and graph levels. Halide separates algorithms from schedules (Ragan-Kelley et al., 2013), while Tensor Comprehensions and TensorIR expose schedule-oriented tensor abstractions (Vasilache et al., 2018; Feng et al., 2023). TVM, AutoTVM, and Ansor combine tensor-program generation with learned search and task scheduling (Chen et al., 2018a,b; Zheng et al., 2020); Triton and OpenTuner cover hand-written GPU programs and extensible autotuning

Table 3: KernelBench Level-2 ablation against PyTorch eager and `torch.compile`. Pass, Fast1, and Fast2 are reported as percentages; Mean, GeoMean, P50, and P75 are speedups.

		vs. PyTorch eager							vs. `torch.compile`					
Model	**Method**	**Pass**	**Fast1**	**Fast2**	**Mean**	**GeoMean**	**P50**	**P75**	**Fast1**	**Fast2**	**Mean**	**GeoMean**	**P50**	**P75**
GPT-5.4	Sample-1	46	4	1	0.693	0.555	0.511	0.640	2	1	0.643	0.507	0.500	0.562
	Sample-10	75	63	8	1.417	1.230	1.064	1.336	44	9	1.344	1.124	1.009	1.323
	Iter-10	100	64	4	1.171	0.999	1.011	1.220	41	5	1.065	0.887	0.980	1.144
	LLM4LLM w/o restart (10)	100	89	32	6.782	1.961	1.250	4.221	66	31	6.573	1.742	1.143	3.986
	LLM4LLM (10)	100	91	32	14.962	2.005	1.332	4.110	71	31	14.716	1.781	1.230	4.020
	LLM4LLM w/o restart (15)	100	90	32	7.026	1.973	1.255	4.221	67	31	6.813	1.752	1.143	3.986
	LLM4LLM (15)	100	95	35	15.929	2.153	1.437	4.691	74	35	15.665	1.912	1.257	4.351
Claude Sonnet 4.6	Sample-1	63	35	4	1.663	1.079	1.009	1.235	29	4	1.548	0.962	0.990	1.100
	Sample-10	90	69	9	2.117	1.302	1.097	1.414	51	11	2.018	1.179	1.020	1.291
	Iter-10	100	62	4	1.183	1.063	1.036	1.342	43	5	1.069	0.944	0.988	1.218
	LLM4LLM w/o restart (10)	100	93	35	17.346	2.267	1.497	4.903	73	35	17.058	2.013	1.299	4.516
	LLM4LLM (10)	99	98	39	28.114	2.447	1.672	4.537	76	39	27.747	2.171	1.379	4.542
	LLM4LLM w/o restart (15)	100	93	38	17.614	2.417	1.591	5.164	76	38	17.317	2.146	1.436	5.117
	LLM4LLM (15)	100	99	41	28.002	2.546	1.683	5.017	77	42	27.638	2.261	1.428	5.051
GLM-5	Sample-1	58	37	14	2.272	1.341	1.196	1.986	31	17	2.191	1.219	1.022	2.333
	Sample-10	87	76	30	4.962	1.953	1.439	3.748	59	33	4.813	1.751	1.286	3.594
	Iter-10	100	70	25	2.207	1.336	1.218	1.999	55	26	2.091	1.187	1.049	2.081
	LLM4LLM w/o restart (10)	97	86	36	16.014	2.179	1.526	4.412	67	35	15.696	1.935	1.303	4.328
	LLM4LLM (10)	100	96	36	20.501	2.539	1.555	5.113	76	37	23.199	2.255	1.333	5.181
	LLM4LLM w/o restart (15)	99	94	37	16.902	2.294	1.590	4.581	71	36	16.553	2.020	1.303	4.482
	LLM4LLM (15)	100	97	39	22.157	2.745	1.680	5.539	76	40	24.718	2.437	1.434	5.281

(Tillet et al., 2019; Ansel et al., 2014). Graph- and model-level systems optimize substitutions, fusion, and dynamic execution across larger computation regions (Jia et al., 2019a,b; Ma et al., 2020; Ansel et al., 2024). LLM4LLM accepts generated kernels through the patched model under a deployment workload.

5.2 LLM Agents for Code and Kernel Generation

Large language models have been applied to code synthesis, repair, feedback-driven improvement, and repository-level editing. Codex, CodeT5, InCoder, and Code Llama establish generation, identifier-aware pretraining, and infilling foundations (Chen et al., 2021; Wang et al., 2021; Fried et al., 2022; Rozière et al., 2024). Search and feedback improve programs in competitive programming, repository repair, and self-refinement settings (Li et al., 2022; Le et al., 2022; Jimenez et al., 2023; Madaan et al., 2023). Reflexion, ReAct, self-debugging, and LLM compiler models connect reasoning traces with execution feedback (Shinn et al., 2023; Yao et al., 2023; Chen et al., 2023; Cummins et al., 2024). KernelBench focuses this capability on GPU kernels (Ouyang et al., 2025); recent systems add reinforcement learning and multi-agent planning (Li et al., 2025; Sun et al., 2026; Dong et al., 2025; Zhu et al., 2025). Memory and verification further address cross-task reuse and benchmark reliability (Dong et al., 2026; Lange et al., 2025). LLM4LLM places the search loop inside a deployment-aware pipeline conditioned on the target inference script.

5.3 Language Model Inference and Serving Optimization

Language-model serving systems optimize memory, batching, scheduling, and execution phases. ORCA and vLLM target iteration scheduling and KV-cache management (Yu et al., 2022; Kwon et al., 2023), while Sarathi and SGLang study chunked execution and structured generation runtimes (Agrawal et al., 2023; Zheng et al., 2024). FlexGen, DeepSpeed Inference, and DeepSpeed-FastGen address memory- and throughput-oriented generation at larger scales (Sheng et al., 2023; Aminabadi et al., 2022; Holmes et al., 2024). At the kernel level, FlashAttention/FlashAttention-2 and FlashInfer provide optimized attention paths for inference workloads (Dao et al., 2022; Dao, 2023; Ye et al., 2025). State-space and hybrid recurrent models add fast paths through Mamba, Mamba-2, causal depthwise convolution, and RecurrentGemma-style recurrent blocks (Gu and Dao, 2023; Dao and Gu, 2024; Dao, 2024; Botev et al., 2024). LLM4LLM uses this deployment context as the environment in which generated candidates are validated and accepted.

6 Conclusion

LLM4LLM reframes LLM-based kernel optimization as a deployment-aware closed-loop problem. The framework connects real-model profiling, phase-aware task construction, experience-guided episodic search, and deployment-time acceptance. By making the target workload part of candidate generation and acceptance, LLM4LLM

aligns kernel search with the execution context that determines inference performance. Across diverse language-model families and two GPU platforms, this loop converts generated candidates into end-to-end inference gains while clarifying the relationship between isolated benchmark performance and deployment behavior. The results suggest a practical path for deployable LLM-generated kernels: profile the real model, search under phase-aware constraints, and accept candidates through target-runtime validation.

Limitations

LLM4LLM currently targets single-GPU inference workloads and assumes access to a representative inference script. The system studies deployment-aware optimization at the module and model-instance level; tensor parallelism, pipeline parallelism, continuous batching, and multi-tenant serving introduce additional acceptance criteria, including communication cost, scheduler interaction, and batch-level interference. The method specializes patches to observed shape and phase regimes, so deployment settings with substantially different prompts, decode lengths, batching behavior, or model configurations require re-profiling and renewed acceptance checks. Optimization cost is also a practical consideration: search is most attractive when the resulting patch is reused across many inference calls or related model instances. The quality of generated candidates depends on the coding ability of the underlying model, the search budget, and the availability of relevant implementation patterns. Future work can extend deployment-time acceptance to distributed serving systems and richer workload mixtures.

Acknowledgments

We thank the anonymous reviewers and the meta-reviewer for their constructive feedback. This work was supported by the Zhongguancun Academy (Grant No. C20250501). This work was also supported in part by the Shaanxi Key Technology R&D Program under Grant 2024GX-ZDCYL-02-15, in part by the Natural Science Funds for Distinguished Young Scholar of Shaanxi under Grant 2025JC-JCQN-079. This work was also supported by the National Key Laboratory of Advanced Communication Networks (Grant No. FFX26641X006).

References

Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. 2023. Sarathi: Efficient LLM inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369*.

Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, and Yuxiong He. 2022. DeepSpeed-Inference: Enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15.

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

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, and 30 others. 2024. 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*, ASPLOS '24, page 929–947, New York, NY, USA. Association for Computing Machinery.

Aleksandar Botev, Soham De, Samuel L Smith, Anushan Fernando, George-Cristian Muraru, Ruba Haroun, Leonard Berrada, Razvan Pascanu, Pier Giuseppe Sessa, Robert Dadashi, Léonard Hussenot, Johan Ferret, Sertan Girgin, Olivier Bachem, Alek Andreev, Kathleen Kenealy, Thomas Mesnard, Cassidy Hardin, Surya Bhupatiraju, and 43 others. 2024. RecurrentGemma: Moving past transformers for efficient open language models. *Preprint*, arXiv:2404.07839.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018a. TVM: An automated End-to-End optimizing compiler for deep

learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA. USENIX Association.

Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018b. Learning to optimize tensor programs. *Advances in Neural Information Processing Systems*, 31.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. 2024. Meta large language model compiler: Foundation models of compiler optimization. *arXiv preprint arXiv:2407.02524*.

Tri Dao. 2023. FlashAttention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*.

Tri Dao. 2024. causal-conv1d: Causal depthwise conv1d in CUDA with a PyTorch interface. https://github.com/Dao-AILab/causal-conv1d. Accessed: 2026-05-04.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. *Advances in neural information processing systems*, 35:16344–16359.

Tri Dao and Albert Gu. 2024. Transformers are SSMs: Generalized models and efficient algorithms through structured state space duality. In *International Conference on Machine Learning (ICML)*.

Juncheng Dong, Yang Yang, Tao Liu, Yang Wang, Feng Qi, Vahid Tarokh, Kaushik Rangadurai, and Shuang Yang. 2025. STARK: Strategic team of agents for refining kernels. *arXiv preprint arXiv:2510.16996*.

Kris Shengjun Dong, Sahil Modi, Dima Nikiforov, Sana Damani, Edward Lin, Siva Kumar Sastry Hari, and Christos Kozyrakis. 2026. KernelBlaster: Continual cross-task CUDA optimization via memory-augmented in-context reinforcement learning. *arXiv preprint arXiv:2602.14293*.

Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, and Tianqi Chen. 2023. TensorIR: An abstraction for automatic tensorized program optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 804–817, New York, NY, USA. Association for Computing Machinery.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. InCoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*.

Albert Gu and Tri Dao. 2023. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*.

Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, and Yuxiong He. 2024. DeepSpeed-FastGen: High-throughput text generation for LLMs via MII and DeepSpeed-Inference. *Preprint*, arXiv:2401.08671.

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

Zhihao Jia, James Thomas, Todd Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2019b. Optimizing dnn computation with relaxed graph substitutions. *Proceedings of Machine Learning and Systems*, 1:27–39.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. SWE-bench: Can language models resolve real-world GitHub issues? *arXiv preprint arXiv:2310.06770*.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626.

Robert Tjarko Lange, Qi Sun, Aaditya Prasad, Maxence Faldor, Yujin Tang, and David Ha. 2025. Towards robust agentic CUDA kernel benchmarking, verification, and optimization. *arXiv preprint arXiv:2509.14279*.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328.

Xiaoya Li, Xiaofei Sun, Albert Wang, Jiwei Li, and Chris Shum. 2025. CUDA-L1: Improving CUDA optimization via contrastive reinforcement learning. *arXiv preprint arXiv:2507.14111*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, and 7 others. 2022. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097.

Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling holistic deep learning compiler optimizations with {rTasks}. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 881–897.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-Refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, pages 46534–46594. Curran Associates, Inc.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. 2025. KernelBench: Can LLMs write efficient GPU kernels? *arXiv preprint arXiv:2502.10517*.

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, and 7 others. 2024. Code Llama: Open foundation models for code. *Preprint*, arXiv:2308.12950.

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. FlexGen: High-throughput generative inference of large language models with a single GPU. In *International Conference on Machine Learning*, pages 31094–31116. PMLR.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652.

Qitong Sun, Jun Han, Tianlin Li, Zhe Tang, Sheng Chen, Fei Yang, Aishan Liu, Xianglong Liu, and Yang Liu. 2026. KernelSkill: A multi-agent framework for GPU kernel optimization. *arXiv preprint arXiv:2603.10085*.

Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 2019. 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*, pages 10–19.

Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 conference on empirical methods in natural language processing*, pages 8696–8708.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.

Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. 2025. FlashInfer: Efficient and customizable attention engine for LLM inference serving. In *Proceedings of Machine Learning and Systems*, volume 7. MLSys.

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX symposium on operating systems design and implementation (OSDI 22)*, pages 521–538.

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. 2020. Ansor: Generating High-Performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 863–879. USENIX Association.

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. 2024. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems*, volume 37, pages 62557–62583. Curran Associates, Inc.

Xinguo Zhu, Shaohui Peng, Jiaming Guo, Yunji Chen, Qi Guo, Yuanbo Wen, Hang Qin, Ruizhi Chen, Qirui Zhou, Ke Gao, Yanjun Wu, Chen Zhao, and Ling Li. 2025. Qimeng-kernel: Macro-thinking micro-coding paradigm for llm-based high-performance gpu kernel generation. *Preprint*, arXiv:2511.20100.

A Appendix

This appendix expands on mechanisms that are only summarized in the main paper. The examples come from saved run artifacts and use the same acceptance principle as the main experiments: a candidate is timed only after it passes the corresponding correctness check. Section A.1 analyzes restart as search-state control; Section A.2 isolates search diversity from linear repair; Section A.3 discusses API compatibility for fast-moving kernel DSLs; and Sections A.4–A.5 summarize model-backbone behavior and accepted deployment kernels.

A.1 Search-State Control by Restart

Table A.1 shows a restart case from KernelBench Level 2, problem 97, `Matmul_BatchNorm_BiasAdd_Divide_Swish`. Both variants use GPT-5.4, Triton fp32, an A100, and a 15-candidate budget. The no-restart run first found a correct epilogue-only candidate, then repeated the same output-mismatch signature for three turns, and eventually returned to a family of correct epilogue-only implementations around 8 ms. The restart-enabled run produced full `Linear + BatchNorm + bias/divide/Swish` fusion inside the GEMM epilogue, reducing the best latency from 8.01 ms to 1.42 ms.

Table A.1: Restart case on KernelBench Level 2 problem 97, measured on A100. The baseline eager runtime is 8.23 ms for both rows.

Variant	Best ms	Speedup
No restart	8.01	1.03×
Restart	1.42	5.80×

This case illustrates the role of restart as search-state control. The useful feedback is compact: the operation sequence, the inference-mode BatchNorm contract, and the need to preserve the bias and activation semantics. The long repair history contains many local choices tied to one partially fused layout. Restart carries forward the compact constraints while allowing the next episode to reselect the optimization scope, which can move the search from epilogue fusion to GEMM-level fusion. The generated code exposes the difference directly:

```
# no restart: best candidate
x = self.matmul(x)
x = self.bn(x)
x = fused_bias_div_swish(
    x, self.bias, self.divide_value)

# restart: best candidate
return fused_linear_bn_bias_div_swish(
    x, self.matmul.weight, self.matmul.bias,
    self.bn.running_mean, self.bn.running_var,
    self.bn.weight, self.bn.bias, self.bias,
    self.bn.eps, self.divide_value)
```

The corresponding kernel bodies show the same distinction:

```
# no restart: epilogue-only kernel
y = (x + b) * inv_divide_value
y = y * tl.sigmoid(y)

# restart: GEMM-level fused kernel
acc += tl.dot(x, w)
acc = acc + lin_b[None, :]
acc = (acc - mean[None, :]) * inv_std[None, :]
acc = acc * gamma[None, :] + beta[None, :]
acc = (acc + extra_b[None, :]) / divide_value
acc = acc * tl.sigmoid(acc)
```

In the no-restart trace, iteration 1 is correct at 8.37 ms, iterations 2–4 repeat the same output-mismatch signature, and iterations 5–15 return to correct epilogue-only variants around 8 ms. With restart enabled, correct full-fusion candidates appear at iteration 2 (1.45 ms) and iteration 3 (1.42 ms), while later candidates explore both fast and slow alternatives.

A.2 Sampling and Iteration Are Complementary

Table A.2 gives two complementary contrasts from GPT-5.4 KernelBench Level 2 runs. Problem 76, `Gemm_Add_ReLU`, shows the value of search diversity: sampling found a full-fusion implementation, while the recorded iterative run stopped at a correct candidate that left GEMM in PyTorch and fused only the bias-ReLU epilogue. Problem 1, `Conv2D_ReLU_BiasAdd`, shows the reverse pattern: the best one-shot sample was correct but slower than eager execution, while iterative repair fixed an initial output mismatch and produced a faster NCHW-specialized epilogue kernel. The point is not that either sampling or iteration dominates; the two mechanisms expose different useful candidates.

Table A.2: Complementary sampling and iteration examples from GPT-5.4 KernelBench Level 2 runs on A100. Speedup is against the eager baseline for each problem.

Problem	Method	Best ms	Speedup
76	Sample-10	1.39	5.76×
76	Iter-10	8.11	0.99×
1	Sample-10	8.12	0.90×
1	Iter-10	6.02	1.21×

For problem 76, the core generated code shows the optimization-scope distinction. The sampled candidate fuses the matrix multiplication and epilogue into one Triton implementation:

```
# Sample-10 best candidate
return fused_linear_bias_relu(x, weight, bias)

# kernel core
acc += tl.dot(a, b)
acc += bias[None, :]
acc = tl.maximum(acc, 0.0)
```

The iterative candidate is correct, but its generated structure preserves the PyTorch GEMM and only moves the epilogue to Triton:

```
# Iter-10 best candidate
y = self.gemm(x)
y = triton_bias_relu(y, self.bias)
return y

# kernel core
y = tl.maximum(x + b[None, :], 0.0)
```

For problem 1, the useful signal is different. The iterative run first produced an output mismatch; after one repair turn, it kept the same conservative module boundary as the sampled code but generated a cleaner NCHW-specialized epilogue kernel:

```
# Sample-10 best candidate
x = self.conv(x)
x = triton_relu_bias(x, self.bias)
return x

# kernel core
hw = H * W
c = (offs // hw) % C
b = tl.load(bias_ptr + c, mask=mask, other=0.0)
y = tl.maximum(x, 0.0) + b

# Iter-10 best candidate, after repair
x = self.conv(x)
x = triton_relu_bias_nchw(x, self.bias)
return x

# kernel core
c = (offs // HW) % C
b = tl.load(bias_ptr + c, mask=mask, other=0.0)
y = tl.maximum(x, 0.0) + b
```

Together, these examples explain why LLM4LLM combines sampling, feedback, and restart instead of treating iterative repair as a purely monotonic process. Sampling exposes alternative decompositions of the same PyTorch graph, while linear repair is effective for turning a nearby candidate into a valid and better-specialized one. Restart combines these roles: each new episode can choose a fresh optimization scope, but it still receives compact correctness and implementation constraints learned from previous attempts.

A.3 API Compatibility and User-Defined Fixes

This section separates interface compatibility from the optimization-scope issue in Section A.2. Kernel DSLs such as Triton and TileLang evolve quickly, and local installations can differ from the APIs seen during model training. In KernelBench Level 2 problem 86, `Matmul_Divide_GELU`, an iterative candidate failed because the generated GELU approximation called an unavailable Triton math entry point:

```
# failed iterative candidate
inner = c0 * (x + c1 * x * x * x)
y = 0.5 * x * (1.0 + tl.math.tanh(inner))
```

After repair, the run obtained a correct epilogue-only kernel, while an independent sampled candidate reached a fused `Linear + divide + GELU` implementation:

Table A.3: API-affected example on KernelBench Level 2 problem 86, measured on A100.

Method	Best ms	Speedup
Sample-10	2.85	2.82×
Iter-10	8.01	1.00×

The generated code differs at the kernel boundary:

```
# Sample-10 best candidate
return triton_linear_div_gelu(
    x, self.linear.weight,
    self.linear.bias, self.divisor)

# Iter-10 repaired candidate
x = self.linear(x)
x = triton_div_gelu(x, self.divisor)
return x
```

LLM4LLM therefore includes an API-correction layer that users can extend for their local backend versions. The correction file is backend specific and currently covers Triton, TileLang, and CUDA extension patterns, for example:

```
tl.math.tanh  -> libdevice.tanh
tl.math.max   -> tl.maximum
tl.math.min   -> tl.minimum
T.Ranged      -> tilelang.language.Range
data<T>()     -> data_ptr<T>()
```

These fixes keep version-dependent interface repair separate from performance-relevant kernel design decisions such as fusion scope, tiling, masking, accumulator precision, cache updates, and launch structure.

A.4 Behavior Across LLM Backbones

The ablation table shows that LLM backbones differ in more than final pass rate. GPT-5.4 follows repair feedback reliably, but the examples above show that it can become conservative once a correct partial-fusion implementation is available. Restart and experience summaries are useful in this setting because they preserve stable constraints without preserving every local edit in the failed trajectory.

Claude Sonnet 4.6 tends to produce well-guarded code with explicit fallback paths. This behavior helps coverage and makes deployment-time acceptance easier to apply, while the search

Table A.4: Representative accepted Triton kernels on extracted deployment tasks, measured on A100. Speedup is measured on the extracted validation task for the module, before full-model aggregation.

Family	Module	Phase / bucket	Main specialization	Task speedup
Transformer	`LlamaAttentionPrefill`	`q_len > 1`	Causal online-softmax attention with fused QKV wiring	2.40×
Transformer	`Qwen3AttentionDecode`	`q_len = 1`	Grouped-query decode sharing each KV tile across query heads	2.01×
Mamba2	`Mamba2MixerPrefill`	prefill scan	Compact recurrent scan with cache-safe mixer wiring	14.31×
Mamba	`MambaMixerDecode`	single-token decode	Fused Conv1d–SiLU–projection chain for cached decode	2.06×
Recurrent	`RecurrentGemmaRglru`	recurrent update	Fused gate normalization plus recurrent-state scan	6.85×

loop still has to test whether guarded candidates enter the optimized path under real prefill/decode conditions. GLM-5 produces more aggressive candidates and higher upper-tail speedups in the KernelBench ablation. The same search loop accommodates these behaviors by using identical correctness, timing, and deployment-acceptance rules for all backbones.

A.5 Deployment Kernel Families

Table A.4 summarizes representative accepted kernels from extracted language-model tasks. The numbers are task-level validation speedups on extracted modules; the end-to-end effect after patching is reported in Table 1 and Figures 5–6.

Transformer attention. The accepted attention kernels are phase specialized. For prefill, the kernel maps one Triton program to a query-head and query-token tile, computes the KV head from the grouped-query structure, embeds causal masking in the score update, and maintains online-softmax statistics in fp32. This structure reduces memory traffic by keeping the attention matrix implicit and by sharing KV loads according to the model's head grouping. For decode, the kernel targets the single-token regime and processes one KV head per program while computing the associated query-head group together. The decode path also keeps mask handling and output layout aligned with the patched module, so the surrounding projection and cache update see the same tensor contract as eager execution.

Mamba-family state-space mixers. The Mamba-family cases show the value of optimizing a deployable module boundary that spans multiple primitive operations. For Mamba2 prefill, the accepted path preserves the mixer sequence of input projection, depthwise convolution, split hidden/state parameters, recurrent scan, gated RMSNorm, and output projection. Inside the recurrent kernel, compact B and C state vectors are loaded once per time step and broadcast logically across heads, while the state tile remains in registers across the scan. For decode, the accepted kernels specialize to `seq_len = 1` with an existing cache, update convolution and SSM state explicitly, and minimize launch overhead for the recurring single-token path. These cases explain why profiling selects Mamba mixers as high-impact targets in Table 1.

RecurrentGemma RGLRU. RecurrentGemma exposes a gated recurrent core whose latency comes from both elementwise gate transformations and recurrent state movement. The accepted RGLRU implementation fuses sigmoid, softplus-derived recurrent gating, reset handling, normalization, and the recurrent scan into Triton kernels that preserve the state update semantics. The key advantage is that the recurrent state is carried through the sequence inside the kernel and the final state is written once, reducing Python-level dispatch and intermediate tensor traffic. This case is distinct from attention and Mamba: the dominant optimization target is the recurrent update itself, which supports the profiling-first design used by LLM4LLM.

Across these families, the accepted kernels are useful because the generated code respects the module boundary that the deployment patch will actually replace. For attention, this means that the code must match the phase-local cache contract: a prefill kernel can assume a query block and construct causal tiles, while a decode kernel must read a populated KV cache and update only the single-token output path. For Mamba and recurrent blocks, the same principle appears as state ownership rather than KV ownership. The kernel must update convolution, SSM, or recurrent state exactly once and return tensors with the same layout expected by the surrounding model. These examples are therefore not just faster isolated kernels. They are accepted because the optimized path can be inserted into the profiled model instance without changing the caller-visible tensor, cache, or phase semantics.

A.6 Summary

The concrete run traces support the design choices in the main paper. Restart is useful when a short set of constraints should survive but a long repair trajectory should not dominate the next implemen-

tation. Sampling provides optimization-scope diversity that a single linear repair path may not expose. API fixes separate version-dependent DSL compatibility from performance-relevant kernel design. The deployment cases show that the accepted kernels are not one generic template: attention, Mamba mixers, and recurrent blocks require different phase contracts and cache semantics.