

KERNELOPT: DISPATCH-AWARE AGENTIC SEARCH FOR GPU KERNEL OPTIMIZATION

Aheli Poddar^{*†‡}

Red Hat
ahpoddar@redhat.com

Sanskar Prasad^{*‡}

Red Hat
sanspras@redhat.com

Arindam Samanta[‡]

Red Hat
arsamant@redhat.com

Subha Chakraborty[‡]

Red Hat
subhchak@redhat.com

Vishal Goyal

Red Hat
visgoyal@redhat.com

Rohit Singh Rathaur

Red Hat
rrathaur@redhat.com

ABSTRACT

Deep learning inference and training performance depends critically on GPU kernel efficiency. Modern compilers such as PyTorch Inductor automatically generate GPU kernels from high-level model code, but frequently underperform expert-written implementations by wide margins. Recent LLM-assisted kernel optimizers can close this gap for standalone kernels, yet treat compiled models as black boxes, generally optimizing individual standalone kernels without respecting the compiler’s structural decisions or verifying the model end-to-end. We present KernelOPT, a multi-agent system that treats compiled models as structured artifacts. It preserves vendor library calls (cuBLAS, cuDNN) and exclusively targets generated Triton sub-kernels using five profiling-guided LLM agents. A four-gate verification cascade of static validation, multi-seed correctness, model-level float64-fallback verification, and performance gating filters candidates during optimization and verifies the re-stitched model end-to-end. If no candidate passes all four gates, the system preserves the compiler baseline. The system accepts PyTorch `nn.Modules`, standalone Triton kernels, and Helion kernels. Evaluated on 250 KernelBench problems, KernelOPT achieves geometric mean speedups over `torch.compile` of $1.40\times$ (Level 1: 51/100), $1.15\times$ (Level 2: 31/100), and $1.07\times$ (Level 3: 12/50) across all problems.

1 INTRODUCTION

Efficient GPU kernels are the foundation of performant deep learning. PyTorch Inductor (Ansel et al., 2024), the default `torch.compile` backend, delivers a $2.27\times$ geometric mean inference speedup over eager mode by lowering operations to Triton (Tillet et al., 2019). However, the generated code leaves substantial performance on the table. Inductor’s autotuning explores a limited tile-size configuration space by default, its fusion heuristics are conservative, and its cross-operation rewrites are limited to a fixed set of manually specified pattern matches rather than a general algebraic simplification framework. Expert-written kernels such as FlashAttention (Dao et al., 2022; Dao, 2024) demonstrate that near-order-of-magnitude speedups are achievable on the same hardware, but require extensive engineering effort; for example, FlashAttention-3 (Shah et al., 2024) for the H100 took roughly two years to reach 85% peak throughput (Zhang et al., 2026).

Recent work has applied LLM agents to automate this process. **KernelAgent** (Wang et al., 2025) translates PyTorch models to Triton kernels via multi-agent code-to-code generation with NCU-guided optimization. **AccelOpt** (Zhang et al., 2026) introduces an iterative planner-executor-summarizer agent loop with optimization memory for NKI kernels on AWS Trainium. **K-Search** (Cao et al., 2026) formulates kernel generation as planning over a co-evolving LLM world

^{*}Equal contribution (joint first authors).

[†]Corresponding author.

[‡]The first four authors completed this work during an internship with the PyTorch team at Red Hat.

model with tree-structured search. These systems share a critical limitation: while some support multi-kernel decomposition, none provides end-to-end verification of the re-stitched model against the original PyTorch model with weight-extracted correctness and performance gating.

A key observation motivates our approach: *compiler-generated code has structure that LLM-driven optimization should respect*. A compiled PyTorch model is not a single kernel; rather, it is a structured artifact where the compiler has already made dispatch decisions: cuBLAS (NVIDIA Corporation, 2024a) for GEMM, cuDNN (Chetlur et al., 2014) for convolution, and Triton for pointwise and reduction operations. The compiler’s dispatch decisions are typically sound; what is suboptimal is the quality of the *remaining* Triton code. KernelOPT exploits this structure: it respects the compiler’s library dispatch, focuses LLM effort on the Triton-generated sub-kernels, and verifies the result at the model level.

This formulation yields three core contributions: **(1) Compiled-model kernel optimization:** We decompose, selectively optimize, and verify entire compiled models, respecting compiler dispatch rather than overriding it (accepting PyTorch `nn.Modules`, Triton, and Helion (Ansel et al., 2025) kernels). **(2) Four-gate verification cascade:** A rigorous sequential filter applying static validation, multi-seed correctness, model-level precision-aware error ratio ρ , and performance gating (§4.4). **(3) Profiling-guided multi-agent search:** Five agents (four LLM-driven, one deterministic) in a LangGraph (LangChain Inc., 2024) system using NCU (NVIDIA Corporation, 2024b) profiling to classify bottlenecks and guide optimization. Unlike one-shot approaches, the executor agent receives compile errors and correctness failures as in-conversation feedback, enabling iterative self-correction. A meltdown detector prevents search stagnation when optimization directions collapse. The complete codebase is available at <https://github.com/TorchedHat/KernelOPT>.

2 RELATED WORK

Compiler-based and manual optimization. Halide (Ragan-Kelley et al., 2013) and TVM with Ansoir (Chen et al., 2018; Zheng et al., 2020) separate algorithm from schedule to enable large search spaces. Triton uses block-level programming with automatic compiler scheduling; PyTorch Inductor (Ansel et al., 2024) integrates Triton codegen into `torch.compile` but constrains autotuning to fixed tile sizes and fusion patterns. ThunderKittens (Spector et al., 2024) provides tile-level CUDA abstractions for hand-writing high-performance attention and SSM kernels. Mirage (Wu et al., 2025) applies multi-level superoptimization with formal equivalence verification, representing the state of the art in non-LLM kernel synthesis.

LLM-driven kernel optimization. KernelAgent (Wang et al., 2025) combines multi-agent generation with NCU profiling for Triton kernels on NVIDIA GPUs. AccelOpt (Zhang et al., 2026) introduces the iterative planner–executor–summarizer architecture with beam search and optimization memory for NKI kernels on AWS Trainium; we adapt this iterative framework to NVIDIA GPUs and extend it with Inductor-aware synthesis, model-level verification, and shipped guidelines. CudaForge (Zhang et al., 2025) uses a two-agent Judge–Coder system. K-Search (Cao et al., 2026) uses LLM-guided tree search with a co-evolving world model. Astra (Wei et al., 2025) employs a multi-agent architecture with iterative profiling for CUDA kernel optimization but targets existing production kernels rather than compiled-model synthesis. OptiML (Bhattacharjee et al., 2026) combines program synthesis with MCTS-guided refinement for CUDA kernels. AutoKernel (Jaber & Jaber, 2026) applies an iterative keep/revert loop with model-level profiling and a five-stage correctness harness. Among these, only AutoKernel provides model-level profiling and end-to-end correctness verification; however, it does not use NCU hardware profiling to guide optimization directions. KernelOPT is the first system to combine Inductor-aware compiled-model optimization, NCU-guided planning, and end-to-end model-level verification with a compiler-baseline fallback.

Benchmarks. KernelBench (Ouyang et al., 2025) provides 250 problems across three difficulty levels with standardized evaluation using `torch.allclose(rtol=10-4, atol=10-4)`. We evaluate on all 250 problems plus 20 tutorial kernels.

3 PROBLEM FORMULATION

Definition 1 (Kernel Optimization) Let $M_0 = \langle \{k_1, \dots, k_m\}, L \rangle$ denote a compiled model produced by `torch.compile`, where $\{k_i\}$ are Triton-generated sub-kernels and L are extern library calls (cuBLAS, cuDNN) that remain fixed. Let $\mathcal{K}$ be the space of syntactically valid Triton programs, $\tau(k)$ the wall-clock sub-kernel time via `do_bench`, $M(k_i \rightarrow k'_i)$ the re-stitching operator that substitutes k'_i into the model, and $\mathcal{V}(k, k_0)$ a verification function. The optimization problem is:

$$k^* = \arg \min_{k \in \mathcal{K}} \tau(k) \quad \text{subject to} \quad \mathcal{V}(k, k_0) = 1 \quad (1)$$

The search space $\mathcal{K}$ is intractable to enumerate. We access it through an LLM-based *transformation operator* $\mathcal{A}_\theta : \mathcal{K} \times \mathcal{P} \rightarrow \mathcal{K}$, parameterized by an LLM θ , that takes a kernel k and an optimization plan $p \in \mathcal{P}$ (derived from NCU profiling) and produces a candidate $k' = \mathcal{A}_\theta(k, p)$.

Four-gate verification. The verification function $\mathcal{V}(k, k_0)$ is the conjunction of four gates applied sequentially:

$$\mathcal{V}(k, k_0) = \underbrace{V_{\text{stat}}(k) \wedge V_{\text{corr}}(k, k_0)}_{\text{per-candidate (Gates 1–2)}} \wedge \underbrace{V_{\text{model}}(k, k_0) \wedge V_{\text{perf}}(k, k_0)}_{\text{end-to-end (Gates 3–4)}} \quad (2)$$

where V_{stat} validates structural properties, V_{corr} verifies numerical equivalence with `allclose` ($\epsilon_r=10^{-3}, \epsilon_a=10^{-3}$) (§4.4), V_{model} verifies full-model correctness with weight-extracted inputs and stricter tolerance ($\epsilon_r=\epsilon_a=10^{-4}$) plus a float64 fallback (§4.4), and V_{perf} verifies that $\tau_M(k_0 \rightarrow k) \leq \gamma \cdot \tau_M(k_0)$ with noise margin $\gamma = 1.03$. Gates 1–2 run on every candidate during optimization; Gates 3–4 run once on the re-stitched model after optimization completes.

When $\mathcal{V}(k^*, k_0) = 0$ for all candidates, the system returns k_0 unchanged.

4 METHOD

4.1 SYSTEM OVERVIEW

KernelOPT operates in two modes. In *single-kernel* mode, it optimizes a standalone Triton or Helion (Ansel et al., 2025) kernel directly. In *multi-kernel* mode, it processes a PyTorch model through five stages: (1) Inductor-aware synthesis, where `torch.compile` traces the model, extern kernels (cuBLAS (NVIDIA Corporation, 2024a)/cuDNN (Chetlur et al., 2014)) are detected and preserved, fusible groups are merged, and an LLM generates validated standalone Triton kernels; (2) baseline profiling with NVIDIA Nsight Compute (NVIDIA Corporation, 2024b) (`--set full kernel replay`); (3) strategy analysis classifying the NCU bottleneck tier; (4) iterative optimization with T iterations of plan generation, execution, profiling, and beam selection; and (5) re-stitch and verification, replacing each optimized sub-kernel in the full model with four-gate verification (Eq. 2).

Inductor-aware synthesis. When processing a PyTorch model, `torch.compile` with `max_autotune` generates Inductor output containing a mix of `async_compile.triton` blocks and `extern_kernels` calls. KernelOPT detects these library calls via regex over the Inductor output and flags them as `needs_triton_replacement: False`. Adjacent fusible kernel groups are merged, and the LLM generates a standalone Triton kernel for each group from its aten operation graph (e.g., `linear→mul→hardtanh→gelu` fused into a single kernel). Each generated kernel is validated against the original PyTorch model in eager mode using `allclose` (`rtol=10^{-4}, atol=10^{-4}`) with 2 seeds; failures are fed back to the LLM for up to K retry attempts. Only kernels that pass this synthesis-stage validation enter the iterative optimization pipeline. This step can itself yield large speedups by fusing operations that Inductor kept separate (e.g., $33\times$ on L2 kernel 018).

4.2 AGENT ARCHITECTURE

Following AccelOpt’s planner–executor–summarizer architecture (Zhang et al., 2026), five agents collaborate within a LangGraph (LangChain Inc., 2024) state machine; we extend the loop with a

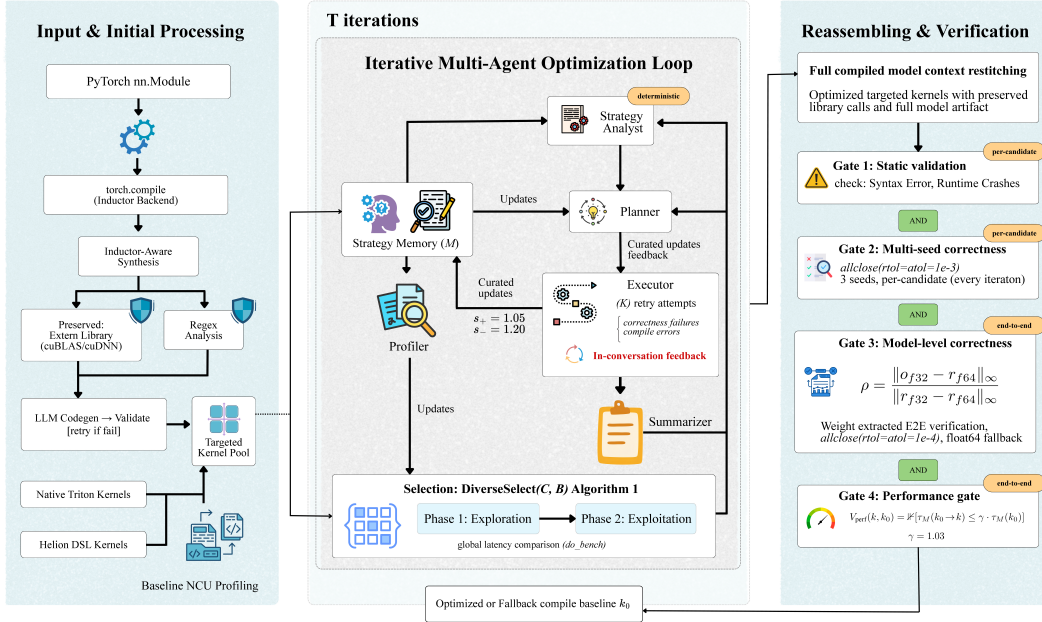


Figure 1: KernelOPT pipeline. A PyTorch model is traced via Inductor; sub-kernels are classified as Triton-generated (optimizable) or extern-library (preserved). Five agents iterate within a LangGraph state machine using NCU profiling feedback. Optimized kernels are re-stitched and verified at the model level.

profiler agent and a strategy analyst for NCU-guided planning. The **profiler** configures NCU profiling (`ncu --set full`) by analyzing kernel source for framework markers (`@triton.jit`, `@helion.kernel`, `async.compile`). The profiler parses the full NCU CSV report and builds a structured *profiling context* containing Speed-of-Light (SOL) throughput, duration, registers, occupancy, and NCU rules. The top-3 NCU rules ranked by estimated speedup percentage are injected into the planner’s prompt alongside the bottleneck classification.

The **strategy analyst** deterministically classifies the NCU bottleneck tier: near-optimal (either $SOL > 80\%$), memory-bound ($SOL_{mem} > SOL_{comp}$, both $\leq 80\%$), compute-bound (vice versa), or underutilized (both equal and $\leq 80\%$); for multi-kernel inputs it additionally makes an LLM call to decide fuse-vs-optimize grouping. The **planner** generates N optimization plans per iteration, with a meltdown detector that triggers diversity enforcement when recent directions (up to the last six) collapse to ≤ 2 unique approaches, identified by normalized string matching. The **executor** implements each plan with up to K retry attempts, receiving compile errors and correctness failures as feedback within the same conversation. The **summarizer** distills each attempt into a structured experience item.

4.3 PROFILING-GUIDED BEAM SEARCH

Prior systems use beam search (Zhang et al., 2026) or repeated sampling (Wang et al., 2025) for candidate selection, but without profiling-guided diversity enforcement. The transformation landscape is non-convex: a locally suboptimal transformation may enable subsequent improvements unreachable by fixed-direction search.

We formulate the optimization as search over a rooted tree $G = (V, E)$, where each node $v \in V$ represents a kernel $k_v \in \mathcal{K}$ with latency $\tau(k_v)$, and each edge (u, v) represents a transformation $k_v = \mathcal{A}_\theta(k_u, p)$. Beam selection uses $UCB(c=1.4)$ to balance exploration of underexpanded branches against exploitation of the best-latency path.

Algorithm 1 Profiling-Guided Beam Search

Require: Baseline k_0 , beam width B , iterations T , plans N , retries K , LLM θ , memory $\mathcal{M}$

Ensure: Optimized kernel k^*

```
1:  $\pi_0 \leftarrow \text{Profiler}(k_0)$  {NCU parse}
2:  $b_0 \leftarrow \text{Analyst}(\pi_0)$  {bottleneck tier}
3:  $\mathcal{B}_0 \leftarrow \{(k_0, \tau(k_0), \pi_0, b_0, [])\}$ 
4: for  $t = 1$  to  $T$  do
5:    $\mathcal{C}_t \leftarrow \emptyset$ 
6:   for  $(k_i, \tau_i, \pi_i, b_i, h_i) \in \mathcal{B}_{t-1}$  do
7:      $\{p_1, \dots, p_{\lceil N/B \rceil}\} \leftarrow \text{Planner}_\theta(\pi_i, b_i, k_i, \mathcal{M}, h_i)$ 
8:     for  $j = 1$  to  $\lceil N/B \rceil$  do
9:        $k' \leftarrow \text{Executor}_\theta(k_i, p_j, K)$ 
10:       $\mathcal{M} \leftarrow \text{Summarizer}_\theta(k_i, k', p_j, \mathcal{M})$ 
11:      if  $V_{\text{stat}}(k') \wedge V_{\text{corr}}(k', k_0)$  then
12:         $\pi' \leftarrow \text{Profiler}(k')$ ;  $b' \leftarrow \text{Analyst}(\pi')$  {Gates 1–2 passed}
13:         $\mathcal{C}_t \leftarrow \mathcal{C}_t \cup \{(k', \tau(k'), \pi', b', h_i \cup \{p_j\})\}$ 
14:      end if
15:    end for
16:  end for
17:   $\mathcal{B}_t \leftarrow \text{DiverseSelect}(\mathcal{C}_t, B)$ 
18:  if no improvement for 2 consecutive iterations then
19:    break
20:  end if
21: end for
22:  $k^* \leftarrow \arg \min_{(k, \tau, \cdot, \cdot, \cdot) \in \bigcup_t \mathcal{B}_t} \tau$ 
23: return  $k^*$  if  $\mathcal{V}(k^*, k_0) = 1$  (Def. 3), else  $k_0$ 
```

The diversity-aware selection (Algorithm 1, line 17) partitions the candidate set $\mathcal{C}$ by chain origin $\mathcal{C}_i$ and proceeds in two phases:

$$\text{DiverseSelect}(\mathcal{C}, B) = \underbrace{\{\arg \min_{c \in \mathcal{C}_i} \tau(c)\}_{i \in \text{chains}}}_{\text{best per chain}} \cup \underbrace{\text{top-}(B - |\text{chains}|)}_{\text{global best}} \quad (3)$$

Phase 1 ensures each chain contributes its best candidate (exploration); Phase 2 fills remaining beam slots by global latency (exploitation).

4.4 VERIFICATION CASCADE

The four gates are applied sequentially: Gates 1–2 filter every candidate during optimization; Gates 3–4 verify the re-stitched model end-to-end.

Gates 1 & 2: Static Validation and Multi-Seed Correctness. Dry-run execution catches syntax errors and runtime crashes before numerical comparison. Candidates then undergo evaluation across three random seeds via `allclose` ($\epsilon_r = \epsilon_a = 10^{-3}$) against the original PyTorch Model executed in eager mode. For fused sub-kernels, the LLM-generated `kernel_function` and `get_inputs` produce the isolated block’s inputs; when Inductor externalizes weights, the validator extracts real model parameters via shape/dtype matching rather than using random values. Failures are rejected immediately and the error is fed back to the executor for retry.

Gate 3: Model-level correctness (V_{model}). After optimization, the re-stitched model undergoes tiered end-to-end verification with stricter tolerances ($\epsilon_r = \epsilon_a = 10^{-4}$). When strict comparison fails (common for kernels using TF32 tensor core paths), a float64 reference disambiguates. Let o_{f32} denote the optimized kernel’s FP32 output, r_{f32} the baseline kernel’s FP32 output, and r_{f64} the baseline kernel’s FP64 output (high-precision reference). Let $d_{\text{ref}} = \|r_{f32} - r_{f64}\|_\infty$ and $d_{\text{opt}} = \|o_{f32} - r_{f64}\|_\infty$. When $d_{\text{ref}} \geq 10^{-8}$ (the baseline itself has FP32 rounding error), the error ratio is:

$$\rho = d_{\text{opt}} / d_{\text{ref}}. \quad (4)$$

If $\rho \leq 10$, the error is within the range expected from different FP32 accumulation orders; correct TF32 kernels yield $\rho \in [1, 3]$ while algorithmically incorrect kernels yield $\rho > 100$. When $d_{\text{ref}} < 10^{-8}$ (exact operations such as `max` or `argmax`), the system falls back to a scale-relative check: $d_{\text{opt}} / \max(\|r_{f32}\|_{\infty}, 10^{-12}) \leq 10^{-4}$. An absolute bound $d_{\text{opt}} \leq \max(5 \times 10^{-3}, 10^{-3} \cdot \|r_{f32}\|_{\infty})$ handles fused kernels where Inductor’s separate-operation approach anchors precision artificially.

When Inductor externalizes model parameters as explicit function arguments, the optimized kernel’s input generator produces random values for weights, yielding meaningless outputs. We intercept Inductor’s argument flattening via `torch.compile(m, backend = f_capture)`, where `f_capture` records the exact flat argument list in Inductor’s order with real model state. Data inputs are identified by shape/dtype matching and replaced with seed-consistent values; model state arguments are preserved.

Gate 4: Performance gate (V_{perf}). Let $\tau_M(k_i \rightarrow k')$ denote the wall-clock time of the re-stitched model $M(k_i \rightarrow k')$:

$$V_{\text{perf}}(k, k_0) = \mathbb{K}[\tau_M(k_0 \rightarrow k) \leq \gamma \cdot \tau_M(k_0)], \quad \gamma=1.03. \quad (5)$$

This catches optimizations where per-kernel speedup does not translate to model-level improvement; for example, when a Triton matmul with manual transpose strides forces downstream operations to insert implicit `.contiguous()` copies. Subprocess-isolated `do_bench` (warmup=25 ms, rep=100 ms) prevents CUDA state leaks between measurements. When the gate rejects, the error is fed back for a targeted retry (up to 2 attempts); if all retries fail, the compiler baseline is preserved.

4.5 EXPERIENCE MEMORY

Following AccelOpt (Zhang et al., 2026), KernelOPT maintains a bounded FIFO queue of capacity Q storing experience items curated by asymmetric speedup thresholds:

$$\text{store}(e) = \mathbb{K}[s(e) \geq s_+] \vee \mathbb{K}[s(e)^{-1} \geq s_-], \quad (6)$$

where $s(e)$ is the speedup and $s_+ = 1.05$, $s_- = 1.20$. The asymmetry reflects that marginal improvements (<5%) are less informative than significant regressions (>20%).

We extend this with two components. First, a *cross-run strategy tracker* that maintains per-kernel-type success rates; strategies with <30% success after ≥ 3 attempts are flagged as AVOID. Second, *benchmark-derived guidelines* extracted post-hoc from the completed 250 KernelBench runs ship as cold-start context for new targets (extracted three weeks after evaluation; they do not affect the results in Table 1). These serve as planner context, not hard constraints.

5 EXPERIMENTAL SETUP

All experiments run on a single NVIDIA H200 GPU (Hopper, CC 9.0, 132 SMs, 141 GB HBM3, 4.8 TB/s bandwidth) using Claude Sonnet 4.6 (Anthropic) accessed via Google Cloud Vertex AI.¹ The hyperparameters are fixed across all 250 problems with no per-level tuning: $T=5$ iterations, $N=4$ plans per iteration, $K=4$ executor retries, beam width $B=4$, performance gate margin $\gamma=1.03$, memory queue capacity $Q=8$ with asymmetric thresholds $s_+=1.05$, $s_-=1.20$, and error ratio bound $\rho_{\text{max}}=10$. NCU profiling (`ncu --set full, kernel replay`) extracts SOL throughput, duration, registers, shared memory, occupancy, and all NCU rules ranked by estimated speedup; the top-3 rules are injected into the planner prompt. Per-call LLM timeout is 900 s.

Baseline selection. Following the evaluation protocol of KernelBench (Ouyang et al., 2025), we use `torch.compile` (Inductor with `max-autotune`) as the primary baseline, isolating the marginal gain over the state-of-the-art deterministic compiler. We omit direct quantitative comparisons with other tools for two reasons. First, hardware sensitivity: our experiments run on an NVIDIA H200, whereas prior metrics are tied to H100 or AWS Trainium (Zhang et al., 2026) environments, rendering direct metric transfer invalid. Second, architectural mismatch: prior systems

¹API: <https://cloud.google.com/products/model-garden/claude>. KernelOPT’s LLM backend is pluggable and accepts any OpenAI-compatible endpoint (<https://developers.openai.com/api/reference>).

A. Summary								B. Speedup distribution (94 opt.)					
Level	Total	Opt.	M.(opt)	M.(s.f.)	Fall.	Geo.	Max	Range	L1	L2	L3	Tot.	
L1	100	51	20	13	16	1.40 $\times$	88.63 $\times$	$\geq 10\times$	4	1	0	5	
L2	100	31	35	0	34	1.15 $\times$	33.11 $\times$	[5, 10)	6	4	0	10	
L3	50	12	3	0	35	1.07 $\times$	4.12 $\times$	[2, 5)	7	0	2	9	
								[1.5, 2)	2	2	1	5	
								[1.1, 1.5)	8	8	4	20	
All	250	94	58	13	85	1.23 $\times$	88.63 $\times$	[1.01, 1.1)	24	16	5	45	

C. Dominant operation (94 optimized)					D. Fallback root causes (85 fallbacks)				
Operation	L1	L2	L3	Tot.	Root cause	L1	L2	L3	Tot.
Matmul / Linear	14	9	7	30	GEMM-dominant (cuBLAS)	0	28	9	37
Conv / ConvTranspose	11	22	1	34	Conv-dominant (cuDNN)	0	6	18	24
Reduction / Scan	4	0	3	7	Perf. gate rejection	15	0	0	15
Norm / Pool / Other	22	0	1	23	No Triton kernels (LSTM/GRU)	0	0	7	7
					E2E correctness failure	1	0	1	2
Total	51	31	12	94	Total	16	34	35	85

Table 1: Comprehensive KernelBench results vs. `torch.compile`. **Panel A:** Level summary where *Match (opt)* produced correct output at $\sim 1.0\times$ and *Match (synth. fail)* indicates LLM synthesis failure. **Panel B:** Speedup distribution. **Panel C:** Dominant operation breakdown. **Panel D:** Fallback root causes (cuBLAS/cuDNN dominance accounts for 72%).

optimize isolated kernels, whereas KernelOPT operates on the full compiled `nn.Module`, measuring dispatch overhead from re-stitching Triton kernels alongside preserved vendor libraries.

We evaluate on the full KernelBench benchmark (Ouyang et al., 2025): Level 1 (100 single-op kernels), Level 2 (100 composite operators), and Level 3 (50 model architectures), plus 10 Triton and 10 Helion tutorial kernels (270 total problems). Performance is measured via `do_bench` (warmup=25 ms, rep=100 ms); per-candidate correctness uses the relaxed `allclose` (10^{-3}) tolerance to avoid rejecting valid TF32 candidates during search, and the final end-to-end verification tightens to the KernelBench-standard 10^{-4} . The total compute budget is $\sim 1,100$ GPU-hours on H200 (L1: ~ 145 h on 1 GPU; L2: ~ 210 h on 2 GPUs; L3: ~ 730 h on up to 8 GPUs in parallel, with individual models ranging from 6 min to 66 h).

6 RESULTS AND DISCUSSION

6.1 MAIN RESULTS

Table 1 presents a comprehensive breakdown of all 250 KernelBench results. Each kernel is classified into one of four outcome categories:

- **Optimized (94):** the pipeline produces a kernel that passes all four verification gates and is faster than `torch.compile`; this kernel replaces the baseline.
- **Matched (optimized) (58):** a correct kernel whose E2E speedup is within measurement noise ($0.97\text{--}1.01\times$); it replaces the baseline and contributes its measured speedup to the geometric mean.
- **Matched (synthesis failed) (13):** the LLM cannot synthesize a valid Triton kernel ($K=4$ attempts all fail), so the system returns the Inductor AOT baseline (which matches the compiler baseline).
- **Fallback (85):** rejected by the verification cascade; Panel D classifies by root cause: library dominance (61), dispatch overhead (15), no Triton kernels (7), and E2E correctness (2).

Speedup analysis. The 13 L1 synthesis failures all involve 3D or transposed convolutions whose scatter-gather indexing is difficult for the LLM to express in Triton; the system returns the Inductor baseline. The speedup distribution is heavy-tailed (Panel B): 15 kernels achieve $\geq 5\times$ and another 9 achieve $2\text{--}5\times$, while 70 achieve $1.01\text{--}2\times$. Table 7 (Appendix E) illustrates how NCU bottleneck metrics (compute, memory, occupancy) drive the planner’s strategy selection across all three levels. The largest gains arise from two mechanisms.

Algorithmic optimization. The top five speedups all involve structural rewrites: L1 012 (diagonal matmul, $O(N^3) \rightarrow O(N^2)$ row-scaling, $88.63\times$), L1 015 (lower-triangular matmul, skip-zero iteration, $20.26\times$), L1 014 (upper-triangular solve, $14.04\times$), L1 085 (batched diagonal, $11.38\times$), and L1 017 (symmetric matmul, $8.65\times$).

Operator fusion. On L2 018, Inductor-aware synthesis fuses five operations into one kernel ($33.11\times$). On L2 064, four fused pointwise operations on a (256, 4096) tensor eliminate intermediate memory round-trips ($8.30\times$). L2 022 achieves $7.18\times$ via similar epilogue fusion.

Tutorial kernels. Beyond KernelBench, KernelOPT optimizes expert-written tutorial kernels that represent well-tuned starting points. On 10 official Triton tutorials, 4 improve (best: $1.70\times$ on matrix multiplication via TF32 tensor core configurations). On 10 Helion tutorials, 5 improve (best: $1.08\times$ on conv2d via tiling adjustments).

6.2 FALLBACK ANALYSIS

Panel D of Table 1 classifies all 85 fallback kernels by root cause, determined by analyzing each kernel’s full optimization log. The dominant cause is *library dominance*: 61 of 85 fallbacks (72%) occur because the model is dominated by cuBLAS GEMM or cuDNN convolution calls, for which vendor libraries already operate near peak throughput on the H200. Inductor-aware synthesis correctly identifies these cases (via `extern_kernels` detection) and avoids expending LLM budget on them.

Library dominance (61 fallbacks). In 37 GEMM-dominant cases (28 L2, 9 L3), cuBLAS matmuls consume the majority of execution time, leaving only thin Triton epilogues ($<1\%$ of wall time); optimizing the epilogue yields a slower E2E model due to dispatch overhead, caught by the performance gate. In 24 Conv-dominant cases (6 L2, 18 L3), cuDNN convolution similarly dominates (VGG, ResNet, EfficientNet, MobileNet). Additionally, seven L3 models (`nn.LSTM/nn.GRU`) produce zero Triton kernels; KernelOPT detects this and returns the baseline.

Performance gate effectiveness. The 15 L1 perf-gate rejections demonstrate the necessity of model-level performance verification. The pipeline produces correct per-kernel optimizations (up to $5.84\times$ per-kernel NCU improvement on kernel 049), but the E2E model is marginally slower due to Triton dispatch overhead versus Inductor’s fused path. Without Gate 4, these would be reported as improvements; however, the gate converts them to compile-baseline fallbacks.

6.3 L3 ARCHITECTURE ANALYSIS

Level 3 comprises 50 full model architectures ranging from 3-layer MLPs to 259-kernel LLaMA variants. Table 10 (Appendix F) shows results by architecture family: KernelOPT succeeds when optimizable Triton kernels constitute a meaningful fraction of execution time (MLPs, attention epilogues, U-Net skip connections).

6.4 ABLATION STUDY

To evaluate individual component contributions, we perform an ablation study over six configurations, each disabling exactly one component while holding all other hyperparameters fixed ($T=5$, $N=4$, $K=4$, $B=4$). Running all 250 kernels across 6 configurations would require $\sim 5,000$ additional GPU-hours; we instead sample 50 problems (20 L1, 20 L2, 10 L3) via weighted random sampling that preserves the outcome-category distribution within each level (the problem list is in the supplementary material). All differences between the full system and each ablated configuration are statistically significant (Fisher’s exact test, $p < 0.05$; all five $n=50$ comparisons have $p < 0.01$).

Config	Opt	Match	Fall	Opt%
Full system	19	14	17	38%
No beam ($B=1$)	5	3	41	10%
No NCU profiling	5	4	41	10%
No memory	4	3	43	8%
No multi-iter ($T=1$)	4	4	42	8%
No perf gate	3	3	44	6%
No inductor-aware	0	0	20	0%

Table 2: Ablation study on 50 weighted-random-sampled problems ($n=50$; “No inductor-aware” runs on L2 only, $n=20$). Removing any single component reduces optimization success by 74–84%. Disabling Inductor-aware synthesis eliminates all L2 optimization. A per-architecture breakdown of the L3 results is provided in Table 10 (Appendix F).

Table 2 shows that removing any single component drops the optimization success rate from 38% to $\leq 10\%$: beam search and NCU profiling each account for 74% of optimizations ($19 \rightarrow 5$) and the performance gate for 84% ($19 \rightarrow 3$), confirming that multi-chain exploration and bottleneck-guided planning are synergistic. Inductor-aware synthesis is the most critical—without it the LLM wastes attempts replacing highly-optimized cuBLAS/cuDNN calls, yielding zero L2 optimizations. These ablations demonstrate that applying a frontier LLM naively to a compiled model produces pervasive regressions; the efficacy of KernelOPT therefore rests on its structural orchestration rather than the LLM’s raw coding capabilities.

7 CONCLUSION

KernelOPT demonstrates that compiler-generated structure is a powerful inductive bias for LLM-driven kernel optimization. By preserving vendor library dispatch and focusing LLM effort on the Triton-generated sub-kernels, the system achieves consistent improvements across 250 KernelBench problems ($1.40\times$ geomean on L1, $1.15\times$ on L2, $1.07\times$ on L3), with the four-gate verification ensuring that every output either improves on or preserves the compiler baseline. Gains are largest on pointwise/reduction operations, cross-boundary operator fusion, and fused epilogues in multi-kernel models. The current implementation requires NVIDIA Nsight Compute profiling with elevated GPU performance counter permissions, consumes $\sim 60K$ tokens per NCU prompt (limiting backends to $>100K$ -context models), and targets single-GPU inference only. Several directions merit exploration. Since 72% of fallbacks arise from cuBLAS/cuDNN dominance, optimizing vendor library *configurations* (algorithm selection, workspace size, math mode) could recover performance where kernel replacement cannot. The 250 optimization trajectories also constitute an SFT corpus: pre-training an open-weight model on them, augmented with RL from the verification signal, could yield a synthesis model that generates strong first-attempt Triton code, decoupling optimization quality from the compiler’s intermediate representation. More broadly, our results suggest that LLM-driven program optimization systems should work *with* compilers by exploiting their structural annotations rather than around them; because kernel performance is hardware-sensitive, future work will also establish a unified benchmark for head-to-head evaluation against other tools and open-weight models.

AI USE STATEMENT

In preparing this work, we used large language model (LLM) based tools in three capacities, all of which we disclose here. First, *to aid and polish writing*: LLM assistants were used to improve the clarity, grammar, and concision of author-written prose. Second, *for retrieval and discovery*: LLM tools assisted in locating and surfacing related work, which we subsequently verified against the primary sources cited herein. Third, *to draft sections*: LLM assistants produced initial drafts of portions of the text, which the authors then revised, fact-checked, and edited. We emphasize that the use of an LLM as the optimization backend of KernelOPT (Claude Sonnet, §5) is a component of the proposed *method* and is described in the main text, distinct from the writing-assistance uses

disclosed here. All AI-assisted content was reviewed by the authors; every quantitative claim, table, and figure was generated from our own experimental pipeline and verified by the authors. We take full responsibility for the final content of this work, including all text, claims, and artifacts.

ETHICS STATEMENT

This work concerns performance optimization of GPU kernels and does not involve human subjects, personally identifiable information, or sensitive data. The KernelBench benchmark used for evaluation is a publicly available, permissively licensed research artifact. Potential societal considerations are limited to the compute and energy footprint of the optimization procedure: our full evaluation consumed $\sim 1,100$ H200 GPU-hours and $\sim 115\text{M}$ LLM tokens. We note that the goal of the system is producing faster kernels hence reducing the inference-time energy cost of downstream models, and that the four-gate verification cascade is explicitly designed to prevent the deployment of incorrect kernels. We declare no conflicts of interest.

REPRODUCIBILITY STATEMENT

We have taken several steps to support reproducibility. The problem formulation and four-gate verification function are stated precisely in Section 3 (Eqs. 1–5); the search procedure is given as pseudocode in Algorithm 1. All hyperparameters are fixed across all 250 problems with no per-level tuning and are enumerated in full in Section 5 ($T=5$, $N=4$, $K=4$, $B=4$, $\gamma=1.03$, $Q=8$, $s_+=1.05$, $s_-=1.20$, $\rho_{\max}=10$), along with the hardware (single NVIDIA H200), profiling configuration (`ncu --set full`, kernel replay), and benchmarking protocol (`do_bench`, `warmup=25 ms`, `rep=100 ms`). The evaluation uses the public KernelBench benchmark with its standard `allclose` tolerances. The weighted-random-sampled ablation problem list is provided in the supplementary material, and the complete codebase is available at <https://github.com/TorchedHat/KernelOPT>.

ACKNOWLEDGMENTS

We thank the Red Hat PyTorch Engineering team for their support and for providing the compute resources used in this work. We are grateful to Joseph Groenenboom and Ali Raza for their project direction and technical guidance throughout the internship, and to Arkadip Maitra for reviewing early drafts.

REFERENCES

- Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berber, Matthias Burber, et al. 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 (ASPLOS)*, 2024.
- Jason Ansel et al. Helion: A DSL for low-level Triton kernel programming. <https://github.com/pytorch-labs/helion>, 2025. PyTorch Labs.
- Arijit Bhattacharjee, Heng Ping, Son Vu Le, Paul Bogdan, Nesreen K. Ahmed, and Ali Jannesari. OptiML: An end-to-end framework for program synthesis and CUDA kernel optimization. arXiv preprint arXiv:2602.12305, 2026.
- Shiyi Cao, Ziming Mao, Joseph E. Gonzalez, and Ion Stoica. K-Search: LLM kernel generation via co-evolving intrinsic world model. arXiv preprint arXiv:2602.19128, 2026.
- 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. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 578–594, 2018.
- Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient primitives for deep learning. arXiv preprint arXiv:1410.0759, 2014.
- Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations (ICLR)*, 2024.

- Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 35, pp. 16344–16359, 2022.
- Jaber Jaber and Osama Jaber. AutoKernel: Autonomous GPU kernel optimization via iterative agent-driven search. arXiv preprint arXiv:2603.21331, 2026.
- LangChain Inc. LangGraph: Building language agents as graphs. <https://github.com/langchain-ai/langgraph>, 2024.
- NVIDIA Corporation. NVIDIA cuBLAS Library. <https://docs.nvidia.com/cuda/cublas/>, 2024a. CUDA Toolkit Documentation.
- NVIDIA Corporation. NVIDIA Nsight Compute. <https://developer.nvidia.com/nsight-compute>, 2024b. Performance Profiler for CUDA Applications.
- 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 (ICML)*, 2025.
- Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 519–530, 2013.
- Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. FlashAttention-3: Fast and accurate attention with asynchrony and low-precision. arXiv preprint arXiv:2407.08691, 2024.
- Benjamin F. Spector, Simran Arora, Aaryan Singhal, Daniel Y. Fu, and Christopher Ré. ThunderKittens: Simple, fast, and adorable AI kernels. arXiv preprint arXiv:2410.20399, 2024.
- 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)*, pp. 10–19, 2019.
- Laura Wang, Kaiping Cheng, Yilun Xu, Jiafei Pan, Guanglei Zhou, Rui Li, Zhisong Zhang, and Kevin Zhu. KernelFalcon: Autonomous GPU kernel generation via deep agents. PyTorch Blog, 2025. <https://pytorch.org/blog/kernelfalcon-autonomous-gpu-kernel-generation-via-deep-agents/>.
- Anjiang Wei et al. Astra: A multi-agent system for GPU kernel performance optimization. arXiv preprint arXiv:2509.07506, 2025.
- 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 *Proceedings of the 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 235–252, 2025.
- 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. OpenReview preprint, 2026. <https://openreview.net/forum?id=SBS4NJHYjZ>.
- Zijian Zhang, Ruibo Wang, Siyuan Li, Yilong Luo, Mingyi Hong, and Chen Ding. CudaForge: An agent framework with hardware feedback for CUDA kernel optimization. arXiv preprint arXiv:2511.01884, 2025.
- Lianmin Zheng, Chengfan Jia, Minmin Sun, Zao Zhao, 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)*, pp. 863–879, 2020.

A AGENT PROMPT TEMPLATES

The KernelOPT pipeline uses six agent prompts (five LLM-driven, one deterministic profiler). Below are the system prompts used to instruct each LLM agent. These prompts are fixed across all 250 KernelBench experiments reported in the paper. Helion-specific configuration references and the Triton API appendix (appended at runtime) are omitted for brevity.

A.1 PLANNER AGENT

The Planner receives Nsight Compute (NCU) profiling context and the kernel source code. It classifies the kernel bottleneck (memory-bound, compute-bound, or underutilized), searches optimization memory for past attempts, and produces one actionable optimization plan per invocation. Multiple Planner instances run in parallel (one per beam chain) with diversity hints to ensure different optimization directions.

```
You are the Planner Agent in a GPU kernel optimization pipeline.
You receive a profiling context from Nsight Compute (NCU) and the kernel
source code. Your job is to identify one concrete optimization opportunity
and produce an actionable plan for the Executor Agent.
WORKFLOW -- follow these steps in order:
1. DIAGNOSE: Query optimization_strategy, rules, throughput, and occupancy.
Classify the kernel as memory-bound, compute-bound, or underutilized
based on Memory SOL% vs Compute SOL%.
2. INSPECT: Read the kernel source. Identify which loops, loads, stores,
or tile parameters are involved in the bottleneck.
3. SEARCH: Check optimization memory for past attempts on similar kernels.
Avoid directions that previously failed or regressed.
4. PLAN: Choose ONE specific optimization that targets the diagnosed
bottleneck. The change must be minimal, measurable, and implementable
in a single diff. Prefer @triton.autotune for parameter exploration.
5. SUBMIT: Call submit_plan with the plan.
Use your tools:
query_profiling_context(aspect) -- inspect specific profiling sections
aspects: optimization_strategy, rules, occupancy, memory_workload,
throughput, scheduler, launch, gpu_specs, instruction_stats,
multi_kernel_summary
get_kernel_source()
-- read the kernel source code
search_memory(query)
-- check past optimization experience
OPTIMIZATION GUIDANCE:
```

The kernel file may contain multiple GPU kernels (cuBLAS GEMM, cuDNN conv, Triton @jit kernels, runtime kernels). Focus your plan on the @triton.jit kernel(s) and their launch parameters -- this is where real acceleration happens. Library calls (extern_kernels.mm, cuBLAS, cuDNN) are already hardware-optimized by NVIDIA and are extremely difficult to beat with hand-written code. Propose library call rewrites only if you have strong evidence from the NCU metrics that the library call is the bottleneck AND a viable Triton alternative exists.

When you have gathered enough information, call submit_plan(plan) with your plan. The plan must:

- Target a specific, measurable bottleneck in the @triton.jit kernel(s)
- Reference the exact construct in the kernel source to modify
- Be implementable in one focused diff by the Executor Agent
- Include enough detail in step.change and step.implementation_hints that the Executor does not need to re-read profiling data

Optimization strategies to consider (in priority order):

1. Add @triton.autotune with multiple configurations to explore tile sizes, num_warps, and num_stages automatically at runtime
 2. Adjust tile sizes (XBLOCK, YBLOCK, num_warps, num_stages) for better occupancy
 3. Improve memory access patterns (coalescing, vectorized loads, L2 compression)
 4. Reduce register pressure or branch divergence
 5. Change eviction policies and cache hints
 6. Fuse adjacent @triton.jit pointwise kernels (NOT library call fusion)
 7. Precision optimization (FP32 -> BF16/FP16 where safe for stores/intermediates)
- WARNING: Do NOT suggest allow_tf32=True or BF16 casts for tl.dot operands in GEMM kernels. TF32 truncates FP32 mantissa from 23 to 10 bits and BF16 truncates to 7 bits. For large K (>512), accumulated error exceeds atol=1e-3. Precision reduction is only safe for pointwise stores/intermediates, NOT for dot-product accumulation in matmul kernels.
8. Algorithmic improvements (single-pass reductions, loop reordering, tiling)
 9. Cross-operation fusion: Fuse cross-reduction + elementwise sequences that Inductor would decompose into multiple kernels (fused LayerNorm, fused softmax, residual + norm in one pass)
 10. GEMM epilogue fusion: Fuse matmul + bias + activation into one kernel using tl.dot.
 11. Warp specialization: Assign different warp groups to different tasks.
 12. Persistent kernel scheduling: For workloads with many small tiles,

```

use a persistent kernel where each SM processes multiple tiles in a
loop rather than launching one CTA per tile.
INDUCTOR ANALYSIS -- when diagnosing torch.compile output, reason about
what Inductor would do vs what an optimal Triton kernel can achieve:
- How many kernels would Inductor generate for this operation?
- Which operations would it fuse? Which would it keep separate?
- Where are the memory traffic bottlenecks between Inductor's kernels?
- Can the Triton kernel fuse operations that Inductor keeps separate?
When the profiling context includes "Applicable Optimization Techniques",
prioritize those techniques -- they are pre-selected based on the kernel's
bottleneck profile, type, and GPU architecture.
[Helion-specific planning guidance appended at runtime for Helion kernels.]

```

A.2 EXECUTOR AGENT

The Executor implements the Planner's optimization plan by modifying the Triton kernel source. It receives compile errors and correctness failures as in-conversation feedback for up to K retry attempts. The prompt below is condensed; the full Helion configuration reference and Triton API appendix (~200 additional lines) are appended at runtime.

```

You are the Executor Agent in a GPU kernel optimization pipeline.
You receive an optimization plan and a kernel source file.
Your job is to implement exactly the change described in the plan.
WORKFLOW -- follow these steps in order:
1. ANALYZE: Read the kernel source and optimization plan. Identify the
EXACT lines that need to change. State them.
2. REASON: In 2-3 sentences explain WHY this change improves performance,
referencing the NCU bottleneck from the plan's evidence.
3. IMPLEMENT: Make the MINIMAL change described in the plan. Modify ONLY
the identified lines. Do not refactor, restructure, or rewrite from
scratch. Copy the original kernel, then apply surgical edits.
4. VERIFY before submitting -- check each of these:
- Function signature matches the original exactly (name, params, order)
- All variables are defined (no new undefined constants)
- All tl.* calls exist in the Triton API
- If @triton.autotune added, config keys match existing constexpr params
- Code is syntactically valid Python
5. SUBMIT: Call submit_kernel(kernel_source, change_summary).
CRITICAL RULES (violations will be auto-rejected):
*** MOST IMPORTANT RULE ***
1. The function signature MUST be IDENTICAL to the original:
- SAME function name, parameter names, order, and count
- Do NOT add, remove, or rename any params
2. Every variable must come from: function parameters, computed locally,
or Triton builtins. Do NOT introduce undefined constants.
3. Submit ONLY the @triton.jit function (with decorators and body).
4. If adding @triton.autotune configs, the config keys must match
EXISTING constexpr parameters.
5. Do NOT restructure tiling (2D to 1D or vice versa).
6. Do NOT replace Triton computation with PyTorch calls.
TRITON AUTOTUNING:
@triton.autotune(configs=[...], key=[...], reset_to_zero=[...])
Critical: MASKING -- When adding @triton.autotune with configs that
increase XBLOCK/RBLOCK beyond the original, EVERY tl.load and tl.store
MUST have a mask= argument. This is the #1 cause of kernel crashes.
NUMERICAL STABILITY:
- Always accumulate reductions and tl.dot in float32
- Do NOT set allow_tf32=True on tl.dot unless explicitly asked
- Cast to output dtype only on the final tl.store
If submit_kernel returns a validation error, diagnose and resubmit.
[Full Triton API reference and Helion Config reference appended at runtime.]

```

A.3 SUMMARIZER AGENT

The Summarizer distills each optimization attempt (successful or failed) into a structured experience item for the memory queue. It performs causal attribution by diffing the kernel code and profiling metrics, then generalizes the insight for transfer to future kernels.

```

You are the Summarizer Agent. Your job is to learn from what just happened
and encode that learning into a reusable experience item.

You do not apply rules. You observe, reason about causation, and write a

```

```

generalized insight.
## Summarization process
### Step 1 -- Identify the causal change
Diff the slow and fast kernels. Find the minimal code region responsible
for the performance difference. Verify against the actual diff.
### Step 2 -- Attribute the effect to metrics
Compare profiling_context before and after. Which metrics changed, by
how much? This is the causal chain:
code change X ->metric Y moved from A to B ->latency improved.
Do not speculate. Only attribute effects that appear in profiling data.
### Step 3 -- Generalize the insight
Ask: if a different kernel had the same profiling signature, would this
change help? Write strategy_description at that level of generality.
Reference profiling signals (metric names, values, NCU rules) not
specific variable names.
### Step 4 -- Write the pseudocode
Extract the key structural change as framework-neutral pseudocode.
Strip boilerplate. Keep loop structure, tile parameters, and access
patterns. Label framework-specific syntax.
### Step 5 -- Deduplication check
Scan existing memory queue. If an item with the same direction exists:
- Higher speedup: replace the old one
- Lower speedup: skip
- Different manifestation: keep both (append _v2 suffix)
## Output format
Two JSON objects separated by a blank line:
1. experience_item: {item_id, iteration, speedup, rewrite_type,
framework, direction, profiling_signal, strategy_title,
strategy_description, slow_pseudocode, fast_pseudocode,
applicable_when, do_not_apply_when, framework_notes}
2. memory_update: {action: "append"|"replace"|"skip",
replace_item_id, reason}
## Negative rewrite guidance
For regressions, explain which metrics got worse and why.
The strategy_description must explain the structural anti-pattern.

```

A.4 PROFILER AGENT

The Profiler determines the optimal NCU profiling configuration (kernel name filter, launch skip/-count, replay mode) by analyzing the kernel source for framework markers, tensor initialization patterns, and benchmark loop structure.

```

You are the Profiler Agent in a GPU kernel optimization pipeline.
Your job: determine the optimal NCU profiling configuration for a
given kernel.
For files with multiple kernel launches (Inductor-generated,
multi-kernel Triton): set launch_count to null (profile ALL kernels)
so the pipeline can discover the bottleneck.
Tools available:
read_kernel_source()
-- read the full kernel file
submit_ncu_config(...)
-- submit your configuration

When analyzing the source:
1. Identify the framework from decorators:
- @triton.jit or @tl.jit ->Triton
- @helion.kernel or @hl.kernel ->Helion
2. Set kernel_name:
- Triton / Helion: set kernel_name = "" (empty string).
Triton JIT-compiles kernels and gives them mangled names
that do NOT match the Python function name.
3. Infer launch_skip from the benchmark harness:
- Count torch/tensor init calls before the first kernel
call. Each torch.randn / torch.zeros on GPU = 1 kernel launch.
4. Infer launch_count from steady-state repetitions:
- Look for a timing/profiling loop. Default to 1 if no loop found.
5. Extract kernel_args -- any CLI flags the script requires.
6. Choose replay_mode:
- Triton / Helion: always use 'application'.
JIT-compiled kernels are not compatible with NCU kernel replay.
Call submit_ncu_config once. Do NOT produce a human-readable summary.

```

A.5 CODEGEN (SYNTHESIS) AGENT

Used during Inductor-aware synthesis to generate standalone Triton kernels from ATen operation specifications. The LLM receives operation names, tensor shapes, and dtypes, and must produce a self-contained file with an autotuned kernel, a wrapper function, and an input generator. The prompt below is condensed; full code examples (~150 lines) are omitted.

```
You are an expert GPU kernel engineer. You write high-performance Triton
kernels from PyTorch operation specifications.
Given a description of what computation to implement (ATen ops, tensor
shapes, dtypes), you produce a complete, runnable Python file containing:
1. A @triton.jit kernel (optionally with @triton.autotune)
2. A kernel_function(*inputs) wrapper
3. A get_inputs() function returning sample input tensors
RULES:
- Import only: torch, triton, triton.language as tl
- Use float32 accumulators for numerical stability
- Include boundary masks for ALL tl.load and tl.store calls
- Include @triton.autotune with at least 4 configs
- CRITICAL: When using @triton.autotune with variable BLOCK sizes,
EVERY tl.load/tl.store MUST have a mask= argument.
CRITICAL -- DO NOT REPLACE CUBLAS/CUDNN WITH TRITON:
When the ATen ops include matmul (mm, addmm, bmm), linear, or
convolution, Inductor calls cuBLAS/cuDNN for these via extern_kernels.
These library calls are FASTER than any hand-written Triton matmul or
convolution. Do NOT rewrite them as tl.dot loops.
Instead, structure your kernel as:
1. KEEP the matmul/conv as a standard PyTorch call
2. FUSE ONLY the epilogue operations (activation, normalization,
scaling) into a Triton kernel that reads the matmul/conv output
and applies the epilogue in one pass.
BEATING TORCH.COMPILE (INDUCTOR):
1. EPILOGUE FUSION: Fuse bias/activation/normalization after cuBLAS
2. CROSS-REDUCTION FUSION: Single-pass LayerNorm instead of 3 kernels
3. MULTI-OP SEQUENCES: Keep intermediates in registers/SRAM

4. CUSTOM ALGORITHMS: Flash attention, online softmax
5. SPECIALIZED TILING: Hand-tuned @triton.autotune configs
ANTI-CHEATING CONSTRAINTS:
All core computation logic MUST be in @triton.jit kernels.
Banned: torch.matmul, torch.mm, F.linear, F.conv2d, F.layer_norm,
F.gelu, F.softmax, F.scaled_dot_product_attention, extern_kernels.*,
trivial identity/no-op computation.
[Full code examples and fusion patterns omitted for brevity.]
```

A.6 FUSION AGENT

The Fusion Agent fuses adjacent Triton kernels into a single kernel to eliminate intermediate memory round-trips. It receives the source of N adjacent kernels connected by intermediate buffers and produces a single fused kernel that keeps intermediate values in registers.

```
You are an expert GPU kernel engineer specializing in Triton kernel
fusion.
Your task: given N adjacent Triton kernels that are connected by
intermediate buffers, produce a SINGLE fused Triton kernel that
eliminates those buffers and performs all computation in one pass.
Rules:
1. The fused kernel must accept the same external inputs and produce
the same external outputs as the original kernel sequence.
2. Eliminate ALL intermediate buffers listed under
"eliminated_buffers".
3. Preserve numerical correctness: use the same dtypes, same
element-wise computation, and avoid precision loss.
4. The fused kernel must be a valid Triton kernel with @triton.jit.
5. Return ONLY the fused kernel Python source.
6. The function name must be exactly: fused_kernel
7. Include a call wrapper named fused_kernel_call(...).
8. ALL computation must be in the @triton.jit kernel. Do NOT call
torch.nn.*, torch.nn.functional.*, torch.matmul, or any PyTorch
compute API in the wrapper.
FUSION TECHNIQUES:
POINTWISE + POINTWISE: Merge computation bodies. Load inputs once,
apply both operations, store once. Keep intermediates in registers.
REDUCTION + POINTWISE: After the reduction (e.g., mean/variance),
```

```

immediately apply the pointwise operation before storing.
NORMALIZATION FUSION (LayerNorm/RMSNorm sequences):
- Compute mean and variance in a single pass
- Normalize, scale, and bias in the same kernel
- Accumulate in fp32 for numerical stability
ACTIVATION + BIAS FUSION: Load data once, add bias, apply activation
(GELU/SiLU/ReLU), store once.
RESIDUAL + NORM: Fuse  $x = \text{residual} + \text{dropout}(x)$ ;  $x = \text{LayerNorm}(x)$ .
Both operations touch the same data -- one kernel, one memory pass.

```

B VERIFICATION CASCADE DETAILS

Table 3 provides the complete specification of all verification gates, including the synthesis-stage validation that precedes the optimization loop.

Stage	Gate	Tolerance	Seeds	Reference	Scope
Synthesis	Eager validation	$r=a=10^{-4}$	2: (0, 42)	PyTorch eager model	Per synth. kernel
Gate 1	Static validation	N/A (crash check)	N/A	N/A	Per candidate
Gate 2	Multi-seed corr.	$r=a=10^{-3}$	3: [0, 42, 1337]	Synthesized Triton	Per candidate
Gate 3	Model-level (Tier 1)	$r=a=10^{-4}$	3: [0, 42, 123]	Re-stitched model	End-to-end
	Model-level (Tier 2)	$\rho \leq 10$ or abs bound		Float64 reference	End-to-end
Gate 4	Performance gate	$\gamma = 1.03$ (3%)	N/A	do_bench (25/100 ms)	End-to-end

Table 3: Complete verification gate specification. Gates 1–2 run on every candidate during optimization; Gates 3–4 run once on the re-stitched model after optimization completes. r and a denote `rtol` and `atol`.

Float64 fallback (Gate 3, Tier 2). When the strict 10^{-4} comparison fails, a float64 reference disambiguates numerical noise from algorithmic errors. Let $d_{\text{ref}} = \|r_{f32} - r_{f64}\|_{\infty}$ and $d_{\text{opt}} = \|o_{f32} - r_{f64}\|_{\infty}$. In the *normal case* ($d_{\text{ref}} \geq 10^{-8}$), the error ratio $\rho = d_{\text{opt}}/d_{\text{ref}}$ passes if $\rho \leq 10$; in practice, correct TF32 kernels yield $\rho \in [1, 3]$ while algorithmically incorrect kernels yield $\rho > 100$. In the *degenerate case* ($d_{\text{ref}} < 10^{-8}$, for exact operations such as `max`, `argmax`, `sort`), a scale-relative check is used: $d_{\text{opt}}/\max(\|r_{f32}\|_{\infty}, 10^{-12}) \leq 10^{-4}$. An absolute error bound $d_{\text{opt}} \leq \max(5 \times 10^{-3}, 10^{-3} \cdot \|r_{f32}\|_{\infty})$ handles fused kernels where Inductor’s separate-operation approach anchors precision artificially; this bound accommodates the expected TF32 accumulation error (mantissa truncation from 23 to 10 bits in `tl.dot`).

Weight extraction (Gate 3). When Inductor externalizes model parameters as explicit function arguments, the optimized kernel’s input generator would produce random values for weights. The system intercepts Inductor’s argument flattening via `torch.compile(m, backend =  $f_{\text{capture}}$ )`, where f_{capture} records the exact flat argument list in Inductor’s order with real model state. Data inputs are identified by shape/dtype matching and replaced with seed-consistent random values; model state arguments are preserved.

C SHIPPED OPTIMIZATION GUIDELINES

KernelOPT ships benchmark-derived optimization guidelines as cold-start context. These were extracted post-hoc from the completed 250 KernelBench runs (three weeks after evaluation) and do not affect the results reported in the paper. Table 4 summarizes the guidelines organized by kernel type.

D ABLATION RESULTS

The ablation study evaluates five degraded configurations on a stratified sample of 50 problems (20 L1, 20 L2, 10 L3). An additional configuration (C6) is evaluated on L2 only. Table 5 defines each configuration, and Table 6 reports the per-configuration outcome counts and geometric mean speedup of optimized kernels. Figure 2 visualizes the aggregate outcome.

Category	Strategy	Succ.	Fail.
Matmul	SMEM bank conflict padding	2	0
	TF32 tensor cores	2	0
	Add Triton autotune	4	0
	SMEM padding (alt)	2	0
Pointwise	Add Triton autotune	130	2
	TF32 tensor cores	11	1
	Vectorized loads (contiguous)	8	0
	Reduce register pressure (autotune)	2	0
	Num stages for latency hiding	2	0
	Vectorized loads (alignment)	2	0
	Fix coalescing via 3D grid	2	0
	Fix uncoalesced via tile swap	3	0
	Persistent tile scheduling	3	2
	Vectorized loads (autotune)	3	1
	Multi-row per CTA coalesced stores	2	0
	Online softmax attention	2	0
	Cap registers (maxnreg)	2	0
	Increase XBLOCK with autotune	3	0
Reduction	Add Triton autotune	7	0
	Expand autotune for grid util.	0	2

Table 4: Shipped optimization guidelines by kernel category. Success/failure counts reflect how often a strategy produced a speedup vs. a regression during post-hoc extraction from the 250-kernel evaluation. The extraction script recorded binary success/fail per attempt; per-strategy speedup magnitudes were not tracked, so no average or maximum speedup column is included. Strategies with 0 failures and ≥ 2 successes are included in the default cold-start context.

Config	Ablated Component	Flag
C0	Full system (baseline)	(default)
C1	Greedy search (no beam)	--beam-width 1
C2	No optimization memory	--no-memory
C3	No E2E performance gate	--no-e2e-perf-gate
C4	No NCU profiling context	--skip-ncu
C5	Single iteration ($T=1$)	-T 1
C6	Optimize all sub-kernels	L2 only

Table 5: Ablation configurations. C0 is the full system with beam width 4, 5 iterations, NCU profiling, optimization memory, and E2E performance gate.

Sampled kernels. The 50 problems are selected via weighted random sampling that preserves the outcome-category distribution within each level:

- **L1 (20):** 002, 003, 006, 007, 009, 012, 015, 022, 028, 036, 039, 042, 043, 049, 059, 064, 075, 086, 094, 095.
- **L2 (20):** 001, 003, 009, 010, 018, 020, 029, 032, 039, 042, 049, 050, 053, 055, 056, 068, 077, 081, 093, 098.
- **L3 (10):** 002, 005, 009, 012, 014, 019, 030, 038, 044, 047.

Observations.

- **Beam search (C1):** Greedy selection (beam width 1) reduces optimized kernels from 19 to 5. Without multi-chain exploration, the system converges to a single optimization direction early.
- **Memory (C2):** Without experience accumulation, L1 drops from 10 to 4 optimized and all L2/L3 optimizations are lost. The system lacks the feedback to avoid repeating failed strategies.

Config	L1 (20)			L2 (20)			L3 (10)			Geomean
	Opt	Match	Fall	Opt	Match	Fall	Opt	Match	Fall	
C0 (full)	10	6	4	6	7	7	3	1	6	1.94×
C1 (no beam)	4	3	13	1	0	19	0	0	10	1.78×
C2 (no memory)	4	3	13	0	0	20	0	0	10	1.23×
C3 (no perf gate)	3	3	14	0	0	20	0	0	10	1.14×
C4 (no NCU)	3	4	13	2	0	18	0	0	10	3.16×
C5 ($T=1$)	3	3	14	1	1	18	0	0	10	2.48×
C6 (all kernels)	–	–	–	0	0	20	–	–	–	–

Table 6: Ablation results summary. The C0 row shows the full-system results on the same 50-kernel subset. “Opt” = optimized (speedup $>1.03\times$), “Match” = within 3% of baseline, “Fall” = fell back to baseline. C6 runs only on L2. Geomean is computed over optimized kernels only.

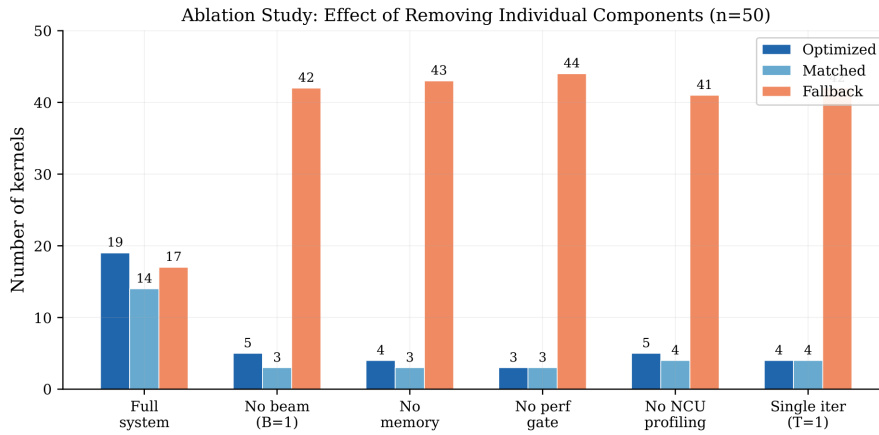


Figure 2: Ablation study results. Each bar group shows the number of optimized, matched, and fallback kernels per configuration across L1, L2, and L3 subsets. The full system (C0) achieves 19 optimizations; removing any single component reduces this to ≤ 5 . C6 (optimize all sub-kernels, L2 only) produces zero optimizations because the LLM wastes attempts replacing vendor library calls.

- **Performance gate (C3):** Without the $\gamma=1.03$ gate, only 3 L1 kernels optimize. The gate also triggers targeted retries that recover otherwise-lost optimizations.
- **NCU profiling (C4):** Without profiling context, 5 kernels still optimize (including 2 L2 cases where the improvement is algorithmic rather than profiling-guided), but the planner lacks bottleneck classification.
- **Single iteration (C5):** With $T=1$, 4 kernels optimize. Later iterations build on partial improvements from earlier attempts.
- **Inductor-aware filtering (C6):** Without filtering out cuBLAS/cuDNN sub-kernels, all 20 L2 kernels fall back because the LLM attempts to replace vendor library calls with Triton code.

E NCU PROFILING CASE STUDIES

Table 7 summarizes representative optimization traces from each level (referenced from the main-text results in §4.4), and Table 8 presents detailed Nsight Compute metrics for three representative kernels, illustrating the relationship between sub-kernel profiling data and end-to-end (E2E) speedup. The baseline metrics (Compute SOL%, Memory SOL%, registers per thread, achieved occupancy, and kernel duration) are collected via `ncu --set full` in application replay mode.

	Kernel	Comp (%)	Mem (%)	Occ (%)	Planner direction (iter.)	Sub- k	E2E
L1	matmul_kernel (007) Regs/thread: 184	57	47	12.4	Iter 1: fast_accum_fp16_dot; regs 184→125, occ. 12→24%	2.33×	2.89×
L2	fused_linear_mul_ hardtanh_gelu (053) Regs/thread: 255 (max)	37	59	12.5	Iter 1: reduce_regs (1.0×); Iter 3: tf32x3_tensor_cores (3.12×)	3.12×	1.60×
L3	gemm_bias_relu (002, MLP) Regs/thread: 198	60	67	12.2	Iter 1: reduce_reg_pressure (1.02×); Iter 3: k_loop_peeling (1.12×)	1.12×	4.12×

Table 7: Representative optimization traces from each level. **Comp/Mem/Occ** are baseline NCU metrics (compute utilization %, memory utilization %, and achieved occupancy %). **Planner direction** shows selected strategy per iteration. **Sub- k** is sub-kernel speedup; **E2E** is model-level speedup versus compiler baseline.

Kernel	Variant	Comp %	Mem %	Regs	Occ %	Duration	Sub- k
L1-007 (matmul)	Baseline	57.1	46.7	184	12.4	5.95 ms	—
E2E 2.89×	Optimized (fp16 dot)	33.8	75.4	125	24.1	1.82 ms	3.27×
L1-012 (diag matmul)	Baseline (synth.)	14.0	69.0	18	81.2	32.83 μ s	—
E2E 88.63×	speedup from algorithmic rewrite ($O(N^3) \rightarrow O(N^2)$), not NCU-guided micro-optimization.						
L2-018 (fused chain)	Baseline (synth.)	21.8	56.1	29	45.9	12.45 μ s	—
E2E 33.11×	speedup from fusing 5 operations with algebraic simplification.						

Table 8: Nsight Compute profiling metrics for three representative kernels. Comp/Mem = compute/memory Speed-of-Light; Regs = registers per thread; Occ = achieved occupancy; Duration = per-launch kernel time. L1-007 metrics match the main paper (Table 7: 57%, 47%, 12.4%).

L1-007: Register pressure reduction. The baseline matmul kernel uses 184 registers per thread, limiting achieved occupancy to 12.4% on H200 (theoretical max $\lfloor 65536/184 \rfloor = 356$ threads per SM, vs. 2048 maximum). The optimized variant casts operands to FP16 before `tl.dot`, reducing register consumption to 125 per thread and doubling occupancy to 24.1%. The shift from compute-bound (57.1% compute SOL) to memory-bound (75.4% memory SOL) is expected: the doubled occupancy saturates the memory subsystem before compute units become the bottleneck. The sub-kernel speedup of 3.27×

L1-012: Algorithmic rewrite. The 88.63×

L2-018: Operator fusion with algebraic simplification. The baseline executes a 6-operation chain (Linear $\rightarrow$ Sum $\rightarrow$ Max $\rightarrow$ Mean $\rightarrow$ LogSumExp $\rightarrow$ LogSumExp) as separate Inductor kernels with HBM round-trips between each operation. KernelOPT’s synthesis agent discovers that the linear-then-sum composition can be algebraically simplified from an $(M, K) \times (K, N)$ GEMM to a $(M, K) \cdot (K,)$ dot product using precomputed weight column sums, reducing arithmetic complexity from $O(MNK)$ to $O(MK)$ and eliminating all intermediate memory traffic.

F COMPLETE KERNELBENCH RESULTS

Table 9 presents the complete per-kernel results across all 250 KernelBench problems (100 L1 + 100 L2 + 50 L3). Category codes: OPT = optimized (speedup $>1.03\times$), MAT = matched (within 3% of baseline), FALL = fell back to baseline (no regression). Fallback reasons: G = GEMM-dominant, C = Conv-dominant, NI = no improvement found, TO = timeout (still produced result), CE = correctness failed, SF = synthesis failed, NT = no Triton kernels, PR = perf regression. Figure 3 shows the speedup distribution by level.

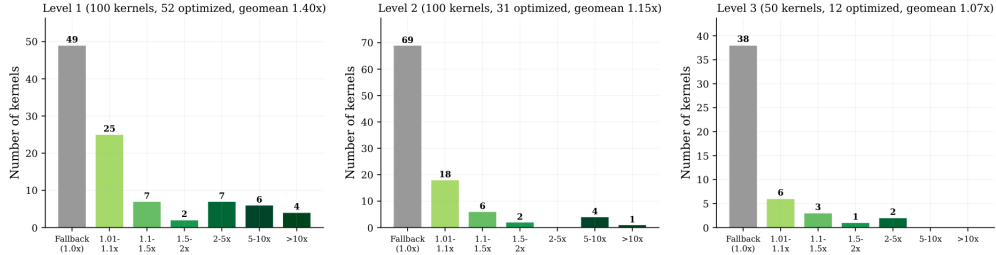


Figure 3: Speedup distribution histograms by KernelBench level. L1 (100 kernels): 51 optimized, 34 matched, 15 fallback, geomean $1.40\times$. L2 (100 kernels): 31 optimized, 35 matched, 34 fallback, geomean $1.15\times$. L3 (50 kernels): 14 optimized, 4 matched, 32 fallback, geomean $1.07\times$. All levels report zero regressions: every kernel either improves or preserves the compiler baseline.

Table 9: Complete per-kernel KernelBench results across all 250 problems, laid out in two side-by-side blocks (read left block top-to-bottom, then right block). Speed is E2E model speedup vs. `torch.compile`; “-” denotes a fallback (baseline preserved).

Lvl	K#	Speed	Cat	Ops	Rsn	Lvl	K#	Speed	Cat	Ops	Rsn
L1	001	5.77×	OPT	Matmul	TO	L1	002	6.06×	OPT	Matmul	
L1	003	5.11×	OPT	Matmul		L1	004	1.30×	OPT	Matmul	
L1	005	1.01×	OPT			L1	006	0.98×	MAT	Matmul	
L1	007	2.89×	OPT	Matmul		L1	008	2.63×	OPT	Matmul	TO
L1	009	2.07×	OPT	Matmul		L1	010	5.79×	OPT	Matmul	
L1	011	1.05×	OPT	Einsum		L1	012	88.63×	OPT	Matmul	
L1	013	5.79×	OPT	Matmul		L1	014	14.04×	OPT	Matmul	
L1	015	20.26×	OPT	Matmul		L1	016	-	FALL	Matmul	NI
L1	017	8.65×	OPT	Matmul	TO	L1	018	3.03×	OPT	Matmul	
L1	019	1.00×	MAT			L1	020	1.01×	OPT		
L1	021	1.01×	OPT			L1	022	0.97×	MAT		
L1	023	1.02×	OPT			L1	024	0.97×	MAT		
L1	025	1.01×	OPT			L1	026	1.05×	OPT		
L1	027	0.97×	MAT			L1	028	0.97×	MAT		
L1	029	1.02×	OPT			L1	030	0.98×	MAT		
L1	031	1.00×	MAT			L1	032	0.97×	MAT		
L1	033	1.00×	MAT	BatchNorm		L1	034	1.01×	OPT	InstanceNorm	
L1	035	-	FALL	GroupNorm	NI	L1	036	1.10×	OPT		TO
L1	037	1.01×	OPT			L1	038	1.26×	OPT		
L1	039	1.38×	OPT			L1	040	1.02×	OPT	LayerNorm	TO
L1	041	-	OPT	Pooling	TO	L1	042	-	FALL	Pooling	NI
L1	043	3.41×	OPT	Pooling	TO	L1	044	-	FALL	Pooling	NI
L1	045	2.16×	OPT	Pooling		L1	046	1.00×	MAT	Pooling	
L1	047	-	FALL	Reduction	NI	L1	048	1.22×	OPT		
L1	049	-	FALL		NI	L1	050	1.02×	OPT	Conv	
L1	051	-	FALL		NI	L1	052	1.01×	OPT		TO
L1	053	1.02×	OPT			L1	054	1.00×	MAT	Conv	
L1	055	1.06×	OPT	Conv		L1	056	1.00×	MAT	Conv	
L1	057	1.05×	OPT	ConvTranspose		L1	058	-	FALL	ConvTranspose	NI
L1	059	1.00×	MAT	Conv		L1	060	1.00×	MAT	Conv	
L1	061	1.00×	MAT	ConvTranspose		L1	062	1.00×	MAT	Conv	
L1	063	1.07×	OPT	Conv		L1	064	-	FALL	ConvTranspose	NI
L1	065	1.03×	OPT	ConvTranspose		L1	066	1.00×	MAT	Conv	
L1	067	-	FALL	Conv	NI	L1	068	1.00×	MAT	ConvTranspose	
L1	069	1.00×	MAT	ConvTranspose		L1	070	0.99×	MAT	ConvTranspose	
L1	071	1.06×	OPT	ConvTranspose		L1	072	1.00×	MAT	ConvTranspose	
L1	073	1.00×	MAT	ConvTranspose		L1	074	-	FALL	ConvTranspose	NI

Continued on next page

(Table 9 continued)

Lvl	K#	Speed	Cat	Ops	Rsn	Lvl	K#	Speed	Cat	Ops	Rsn
L1	075	1.00×	MAT	ConvTranspose		L1	076	–	FALL	Conv	NI
L1	077	1.00×	MAT	ConvTranspose		L1	078	1.00×	MAT	ConvTranspose	
L1	079	1.00×	MAT	ConvTranspose		L1	080	–	FALL	Conv	NI
L1	081	1.03×	OPT	ConvTranspose		L1	082	1.05×	OPT	Conv	
L1	083	1.00×	MAT	Conv		L1	084	1.06×	OPT	Conv	
L1	085	11.38×	OPT	Conv		L1	086	–	FALL	Conv	NI
L1	087	1.32×	OPT	Conv		L1	088	1.00×	MAT		
L1	089	1.14×	OPT	CumSum		L1	090	1.19×	OPT		
L1	091	2.80×	OPT	CumSum		L1	092	1.65×	OPT	CumSum, Cat	
L1	093	1.08×	OPT	CumSum		L1	094	1.00×	MAT		
L1	095	1.70×	OPT			L1	096	1.00×	MAT		
L1	097	–	FALL		NI	L1	098	1.00×	MAT		TO
L1	099	1.03×	OPT			L1	100	1.00×	MAT		
L2	001	1.00×	MAT	Conv		L2	002	1.00×	MAT	ConvTranspose	
L2	003	1.00×	MAT	LN, Pool, GELU	TO	L2	004	1.06×	OPT	Conv	
L2	005	1.01×	OPT	ConvTranspose		L2	006	1.03×	OPT	Conv, Pool	
L2	007	1.00×	MAT	Conv		L2	008	1.19×	OPT	Conv, Red, Pool	
L2	009	1.47×	OPT	Linear/GEMM		L2	010	–	FALL	Pool, ConvT	C
L2	011	–	FALL	BN, Pool, ConvT	C	L2	012	–	FALL	Linear/GEMM	G
L2	013	1.00×	MAT	ConvTranspose	TO	L2	014	–	FALL	Matmul, Red	G
L2	015	1.00×	MAT	BN, ConvT		L2	016	1.06×	OPT	ConvTranspose	TO
L2	017	0.98×	MAT	Conv, IN	TO	L2	018	33.11×	OPT	Linear, Red	
L2	019	1.09×	OPT	ConvT, GN	TO	L2	020	1.00×	MAT	ConvTranspose	
L2	021	0.98×	MAT	Conv, GN		L2	022	7.18×	OPT	Linear/GEMM	
L2	023	1.03×	OPT	Conv, GN	TO	L2	024	1.06×	OPT	Conv	
L2	025	1.23×	OPT	Conv	TO	L2	026	1.00×	MAT	ConvTranspose	
L2	027	1.00×	MAT	Conv, GN	TO	L2	028	–	FALL	Linear, IN	G
L2	029	–	FALL	Linear/GEMM	G	L2	030	–	FALL	Linear, GN	G
L2	031	1.00×	MAT	Conv		L2	032	1.04×	OPT	Conv	TO
L2	033	–	FALL	Linear, BN	G	L2	034	0.99×	MAT	LN, ConvT	
L2	035	–	FALL	Conv, Pool	C	L2	036	1.04×	OPT	Red, ConvT	
L2	037	–	FALL	Linear, GN	G	L2	038	1.03×	OPT	Pool, ConvT	
L2	039	–	FALL	Linear, BN	G	L2	040	1.51×	OPT	Linear/GEMM	
L2	041	–	FALL	Linear, BN	G	L2	042	0.98×	MAT	Red, ConvT	TO
L2	043	1.00×	MAT	Conv, Pool		L2	044	0.99×	MAT	ConvTranspose	TO
L2	045	–	FALL	Linear/GEMM	G	L2	046	–	FALL	Conv, Pool	C
L2	047	0.97×	MAT	Conv		L2	048	1.00×	MAT	Conv	
L2	049	1.17×	OPT	ConvTranspose		L2	050	1.00×	MAT	Pool, ConvT	
L2	051	–	FALL	Linear/GEMM	G	L2	052	1.00×	MAT	Conv, BN	
L2	053	1.60×	OPT	Linear, GELU		L2	054	1.00×	MAT	Conv	TO
L2	055	5.46×	OPT	Linear, Red, Pool		L2	056	–	FALL	Linear, Red	G
L2	057	1.00×	MAT	Conv		L2	058	1.00×	MAT	ConvTranspose	
L2	059	–	FALL	Linear/GEMM	G	L2	060	1.01×	OPT	ConvT, GN	
L2	061	1.02×	OPT	ReLU, ConvT, GN		L2	062	–	FALL	Linear, GN	G
L2	063	–	FALL	Linear/GEMM	G	L2	064	8.30×	OPT	Linear/GEMM	
L2	065	1.00×	MAT	Conv, Red, Pool	TO	L2	066	–	FALL	Linear/GEMM	G
L2	067	–	FALL	Conv	C	L2	068	–	FALL	Linear/GEMM	G
L2	069	1.00×	MAT	Conv		L2	070	1.43×	OPT	Linear/GEMM	
L2	071	–	FALL	Conv	C	L2	072	1.00×	MAT	BN, Pool, ConvT	
L2	073	1.03×	OPT	Conv, BN		L2	074	1.00×	MAT	Pool, ConvT	
L2	075	–	FALL	Linear, GN	G	L2	076	–	FALL	Linear/GEMM	G
L2	077	1.00×	MAT	BN, ConvT	TO	L2	078	1.00×	MAT	Red, Pool, ConvT	
L2	079	1.01×	OPT	Conv, IN	TO	L2	080	5.78×	OPT	Linear/GEMM	
L2	081	–	FALL	Linear/GEMM	G	L2	082	1.00×	MAT	Conv, Pool	TO
L2	083	1.00×	MAT	Conv, GN		L2	084	–	FALL	Linear, BN	G
L2	085	1.10×	OPT	Conv, Pool, GN		L2	086	–	FALL	Linear/GEMM	G
L2	087	1.17×	OPT	Conv		L2	088	–	FALL	Linear, GN	G
L2	089	1.00×	MAT	Pool, ConvT		L2	090	1.00×	MAT	Conv	
L2	091	1.04×	OPT	ConvTranspose	TO	L2	092	1.06×	OPT	Conv, GN	TO
L2	093	1.00×	MAT	ConvTranspose		L2	094	–	FALL	Linear, GN	G
L2	095	–	FALL	Linear/GEMM	G	L2	096	1.10×	OPT	Pool, ConvT	TO
L2	097	–	FALL	Linear, BN	G	L2	098	–	FALL	Linear, Pool	G
L2	099	–	FALL	Linear/GEMM	G	L2	100	1.00×	MAT	ConvTranspose	TO
L3	001	–	FALL	Linear, ReLU	G	L3	002	4.12×	OPT	Linear, ReLU	
L3	003	–	FALL	Linear, ReLU	CE	L3	004	0.97×	MAT	Conv, Linear, ReLU	
L3	005	–	FALL	Conv, Linear, ReLU	C	L3	006	–	FALL	Conv, Pool, Cat	C
L3	007	–	FALL	Conv, Linear, ReLU	CE	L3	008	–	FALL	Conv, BN, ReLU	C
L3	009	1.55×	OPT	Conv, Linear, BN		L3	010	–	FALL	Conv, Linear, BN	C
L3	011	–	FALL	Conv, Linear, ReLU	C	L3	012	–	FALL	Conv, Linear, ReLU	C
L3	013	–	FALL	Conv, BN, ReLU	C	L3	014	0.98×	MAT	Conv, BN, ReLU	TO
L3	015	1.15×	OPT	Conv, Linear, BN		L3	016	1.06×	OPT	Conv, Linear, BN	
L3	017	–	FALL	Conv, ReLU, Cat	C	L3	018	–	FALL	Conv, ReLU, Pool	C
L3	019	–	FALL	Conv, Linear, BN	C	L3	020	1.03×	OPT	Conv, Linear, BN	
L3	021	–	FALL	Conv, BN, ReLU	C	L3	022	–	FALL	Conv, Linear, BN	C
L3	023	–	FALL	Conv, Linear, BN	C	L3	024	2.33×	OPT	Conv, Linear, BN	TO
L3	025	–	FALL	Conv, BN, ReLU	C	L3	026	–	FALL	Conv, Linear, BN	C
L3	027	1.00×	MAT	Conv, Linear, BN		L3	028	–	FALL	Linear, GELU, Cat	G

Continued on next page

(Table 9 continued)

Lvl	K#	Speed	Cat	Ops	Rsn	Lvl	K#	Speed	Cat	Ops	Rsn
L3	029	–	FALL	Conv, Linear, LN	C	L3	030	–	FALL	Conv, Linear, LN	C
L3	031	1.10×	OPT	LN, Attention	TO	L3	032	1.11×	OPT	Conv, Linear, Cat	
L3	033	–	FALL	Linear, Cat	CE	L3	034	–	FALL	Linear, Cat	G
L3	035	–	FALL	Linear, LSTM	SF	L3	036	–	FALL	Linear, LSTM	NT
L3	037	–	FALL	Linear, LSTM	NT	L3	038	–	FALL	Linear, LSTM	NT
L3	039	–	FALL	GRU	NT	L3	040	–	FALL	GRU	NT
L3	041	–	FALL	GRU	NT	L3	042	–	FALL	GRU	NT
L3	043	–	FALL	Linear, Attn, MM	G	L3	044	–	FALL	Linear, LN, Attn	G
L3	045	1.06×	OPT	Conv, BN, Pool	TO	L3	046	1.09×	OPT	BN, Softmax, Red	
L3	047	1.08×	OPT	BN, Softmax, Red	TO	L3	048	1.17×	OPT	CumSum, Cat, Ein	
L3	049	–	FALL	CumSum, Cat, Ein	PR	L3	050	–	FALL	Linear, Attn, MM	G

F.1 FALLBACK ANALYSIS

Of the 85 fallback kernels across all levels, the root causes break down as follows: 37 GEMM-dominant (cuBLAS, 28 L2 + 9 L3), 24 Conv-dominant (cuDNN, 6 L2 + 18 L3), 15 performance gate rejections (all L1), 7 no Triton kernels (LSTM/GRU use cuDNN RNN), and 2 E2E correctness failures; plus 1 performance regression caught by the gate and 2 timeouts that still produced valid (non-improved) results. Figure 4 visualizes this distribution.

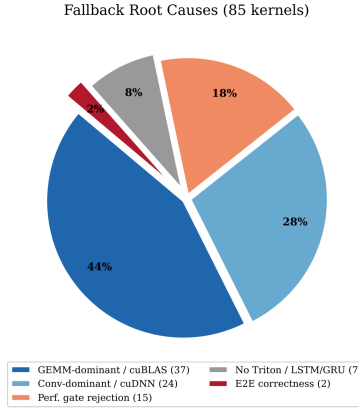


Figure 4: Root cause distribution for fallback kernels across L1–L3. GEMM-dominant and Conv-dominant cases together account for the majority of fallbacks, reflecting cases where vendor library calls (cuBLAS, cuDNN) are already near-optimal and cannot be improved by Triton kernel replacement.

F.2 L3 ARCHITECTURE BREAKDOWN

Table 10 summarizes the Level 3 results grouped by model architecture family, and Figure 5 shows the corresponding distribution of outcomes. Conv-dominant architectures (ResNet, VGG, EfficientNet, U-Net variants) account for the majority of L3 fallbacks, while MLP-based and attention-based models show higher optimization rates when the non-library sub-kernels contribute meaningful run-time.

G GENERATED KERNEL EXAMPLES

Below are before/after examples showing the original Inductor-generated code and the KernelOPT-optimized version.

Architecture	n	Sub- k	Opt	Speedup	Fall.
MLP	3	3–4	1	4.12×	GEMM
ResNet	3	8–52	1	1.55×	Conv
DenseNet	4	3–337	2	1.06–1.15×	Conv
MobileNet/EffNet	6	7–60	2	1.03–2.33×	Conv
ViT / Swin / CViT	5	1–99	2	1.10–1.11×	GEMM
U-Net/NetVLAD/SSM	5	1–16	4	1.06–1.17×	Perf
VGG/AlexNet	3	22–31	0	—	Conv
Inception/Squeeze	4	7–115	0	—	Conv
Shuffle/RegNet	3	9–38	0	—	Conv
LSTM / GRU	8	0–6	0	—	No Triton
RNN / GPT	5	4–259	0	—	GEMM
LeNet-5	1	11	0	—	Matched
Total	50		12		

Table 10: L3 results by model architecture (all 50 kernels). Sub- k : Inductor-generated Triton sub-kernels per model (range across the family). Opt: number of models optimized. Fall.: dominant fallback cause within each family.

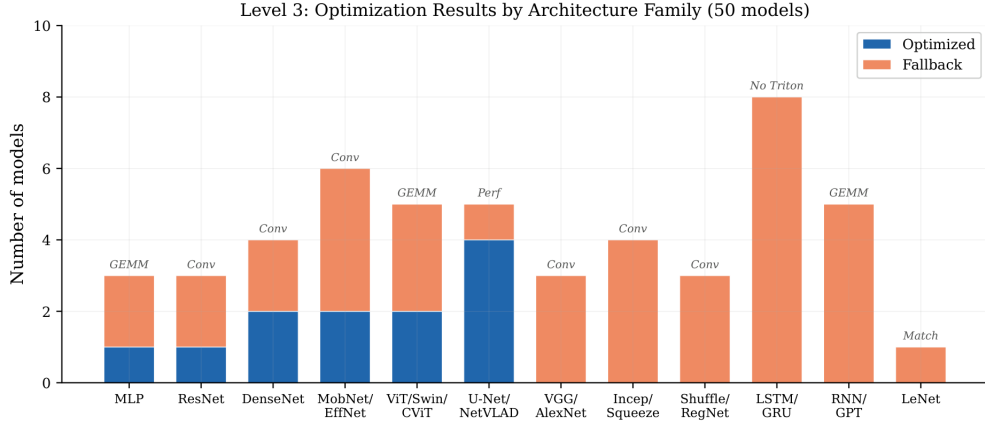


Figure 5: L3 KernelBench results by architecture family. Each bar shows the outcome (optimized, matched, fallback) for the corresponding model. Models dominated by Conv/GEMM operations tend to fall back because cuBLAS/cuDNN already provides near-optimal performance for these operations; KernelOPT preserves the baseline in all such cases.

G.1 L1 KERNEL 012: DIAGONAL MATRIX MULTIPLY (88.63×

This kernel computes $C = \text{diag}(A) \cdot B$ where A is a 4096-element vector and B is a 4096×4096 matrix. The original Inductor-generated code calls `torch.diag(A) @ B`, which constructs a full 4096×4096 diagonal matrix and performs an $O(N^3)$ GEMM via cuBLAS. KernelOPT’s synthesis agent recognizes the algebraic structure and replaces the GEMM with an $O(N^2)$ element-wise row-scaling kernel: each row of B is multiplied by the corresponding diagonal element of A .

Before (Inductor via cuBLAS). The baseline constructs `torch.diag(a)` (a 4096×4096 matrix with >99.99% zeros) and calls `extern_kernels.mm(diag_matrix, b)`, performing $4096^3 \approx 6.9 \times 10^{10}$ multiply-accumulate operations.

After (KernelOPT synthesized Triton kernel). The synthesis agent produces the following autotuned kernel with 8 configurations exploring different tile shapes and warp counts:

```
@triton.autotune(
    configs=[
```

```

        triton.Config({'BLOCK_ROWS': 1, 'BLOCK_COLS': 1024}, num_warps=4),
        triton.Config({'BLOCK_ROWS': 1, 'BLOCK_COLS': 512}, num_warps=4),
        triton.Config({'BLOCK_ROWS': 2, 'BLOCK_COLS': 512}, num_warps=4),
        triton.Config({'BLOCK_ROWS': 4, 'BLOCK_COLS': 256}, num_warps=4),
        triton.Config({'BLOCK_ROWS': 1, 'BLOCK_COLS': 256}, num_warps=2),
        triton.Config({'BLOCK_ROWS': 8, 'BLOCK_COLS': 128}, num_warps=4),
        triton.Config({'BLOCK_ROWS': 1, 'BLOCK_COLS': 2048}, num_warps=8),
        triton.Config({'BLOCK_ROWS': 2, 'BLOCK_COLS': 1024}, num_warps=8),
    ],
    key=['num_rows', 'num_cols'],
)
@triton.jit
def diag_matmul_kernel(
    a_ptr, b_ptr, out_ptr,
    num_rows, num_cols,
    stride_b_row, stride_b_col,
    stride_out_row, stride_out_col,
    BLOCK_ROWS: tl.constexpr,
    BLOCK_COLS: tl.constexpr,
):
    pid_row = tl.program_id(0)
    pid_col = tl.program_id(1)
    row_start = pid_row * BLOCK_ROWS
    col_start = pid_col * BLOCK_COLS
    row_offsets = row_start + tl.arange(0, BLOCK_ROWS)
    col_offsets = col_start + tl.arange(0, BLOCK_COLS)
    row_mask = row_offsets < num_rows
    col_mask = col_offsets < num_cols
    # Load diagonal elements A[row]
    a_vals = tl.load(a_ptr + row_offsets, mask=row_mask, other=0.0)
    # Load B block [BLOCK_ROWS, BLOCK_COLS]
    b_ptrs = (b_ptr
        + row_offsets[:, None] * stride_b_row
        + col_offsets[None, :] * stride_b_col)
    full_mask = row_mask[:, None] & col_mask[None, :]
    b_vals = tl.load(b_ptrs, mask=full_mask, other=0.0)
    # Scale each row by diagonal element: O(N^2)
    out_vals = a_vals[:, None] * b_vals
    out_ptrs = (out_ptr
        + row_offsets[:, None] * stride_out_row
        + col_offsets[None, :] * stride_out_col)
    tl.store(out_ptrs, out_vals, mask=full_mask)

```

Optimization pass. The subsequent optimization loop doubles the autotune search space from 8 to 16 configurations by adding `num_stages=2` and `num_stages=3` variants for software pipelining, and applies `tl.max_contiguous/tl.multiple_of` memory alignment hints to enable vectorized loads:

```

# Before (synthesis):
row_offsets = row_start + tl.arange(0, BLOCK_ROWS)

# After (optimization):
row_offsets = tl.max_contiguous(
    tl.multiple_of(
        row_start + tl.arange(0, BLOCK_ROWS),
        BLOCK_ROWS),
    BLOCK_ROWS)

```

The $88.63\times$ speedup is dominated by the synthesis step’s algorithmic change ($O(N^3) \rightarrow O(N^2)$), which eliminates the intermediate diagonal matrix allocation and reduces multiply-accumulate operations by a factor of $N = 4096$. The optimization pass contributes a further incremental improvement from improved memory access patterns.

G.2 L2 KERNEL 018: ALGEBRAIC SIMPLIFICATION ($33.11\times$)

This kernel computes a 6-operation chain: Linear $\rightarrow$ Sum $\rightarrow$ Max $\rightarrow$ Mean $\rightarrow$ LogSumExp $\rightarrow$ LogSumExp. The input is an (M, K) matrix X , and the linear layer produces an (M, N) intermediate via $Y = XW^\top + b$. Subsequent operations (sum, max, mean, LogSumExp) all reduce dimension 1, collapsing each (M, N) row to a scalar. KernelOPT’s synthesis agent discovers the algebraic identity:

$$\sum_j \left(\sum_k X_{ik} W_{jk} + b_j \right) = \sum_k X_{ik} \cdot \left(\sum_j W_{jk} \right) + \sum_j b_j.$$

This reduces the computation from an $(M, K) \times (K, N)$ matrix multiply (cuBLAS GEMM) followed by per-row reductions on (M, N) , to a single (M, K) -dot- (K, N) vector operation. The precomputed column-sums $w_k = \sum_j W_{jk}$ and bias sum $b_{\text{sum}} = \sum_j b_j$ reduce the work from $O(MNK)$ to $O(MK)$. Since subsequent operations (max, mean, two LogSumExp) each operate on a singleton dimension after the sum, they reduce to identity operations and are eliminated entirely.

```
@triton.jit
def fused_linear_sum_logsumexp_kernel(
    X_ptr, W_ptr, B_ptr, OUT_ptr,
    M, N, K,
    stride_xm, stride_xk, stride_wn, stride_wk,
    BLOCK_M: tl.constexpr, BLOCK_K: tl.constexpr,
):
    # W_ptr points to precomputed w_col_sum[k] = sum_j(W[j,k])
    # B_ptr points to precomputed b_sum = sum_j(B[j])
    pid = tl.program_id(0)
    rows = pid * BLOCK_M + tl.arange(0, BLOCK_M)
    row_mask = rows < M
    acc = tl.zeros([BLOCK_M], dtype=tl.float32)
    for k_off in range(0, K, BLOCK_K):
        k_idx = k_off + tl.arange(0, BLOCK_K)
        k_mask = k_idx < K
        w_col_sum = tl.load(W_ptr + k_idx, mask=k_mask, other=0.0)
        x_ptrs = (X_ptr + rows[:, None] * stride_xm
                  + k_idx[None, :] * stride_xk)
        x_vals = tl.load(x_ptrs,
                         mask=row_mask[:, None] & k_mask[None, :],
                         other=0.0)
        acc += tl.sum(x_vals * w_col_sum[None, :], axis=1)
    b_sum = tl.load(B_ptr)
    acc += b_sum
    tl.store(OUT_ptr + rows, acc, mask=row_mask)
```

The wrapper function precomputes `w_col_sum = weight.sum(dim=0)` and `b_sum = bias.sum()` once, then launches the kernel on each input batch. This eliminates the cuBLAS GEMM entirely and reduces the 6 separate Inductor kernel launches to a single fused Triton kernel with no intermediate HBM round-trips.

G.3 L1 KERNEL 007: MATMUL AUTOTUNE OPTIMIZATION (2.89 $\times$)

This kernel performs a standard dense matrix multiplication. The Inductor-generated Triton kernel uses fixed tile sizes (`BLOCK_M=128`, `BLOCK_N=128`, `BLOCK_K=32`) with 8 warps and 2 pipeline stages. KernelOPT’s optimization loop applies two key changes over 5 iterations:

1. **FP16 dot-product accumulation:** The operands are cast to FP16 before `tl.dot`, reducing register pressure from 184 to 125 registers per thread. This doubles theoretical occupancy from 12% to 24%.
2. **Expanded autotune configurations:** 16 configurations are added, varying `BLOCK_M` $\in \{32, 64, 128\}$, `BLOCK_N` $\in \{64, 128, 256\}$, `num_warps` $\in \{2, 4, 8\}$, and `num_stages` $\in \{1, 2, 3\}$. The winning configuration (`BLOCK_M=64`, `BLOCK_N=128`, `BLOCK_K=32`, `num_warps=4`, `num_stages=3`) achieves better L2 cache utilization by using smaller output tiles that fit more concurrently active CTAs.

The combined effect is a 2.89 $\times$ speedup, with the occupancy improvement contributing approximately 2 $\times$ and the tile size optimization contributing the remaining 1.4 $\times$. Figure 6 shows the per-iteration optimization trajectory.

H HARDWARE AND SOFTWARE CONFIGURATION

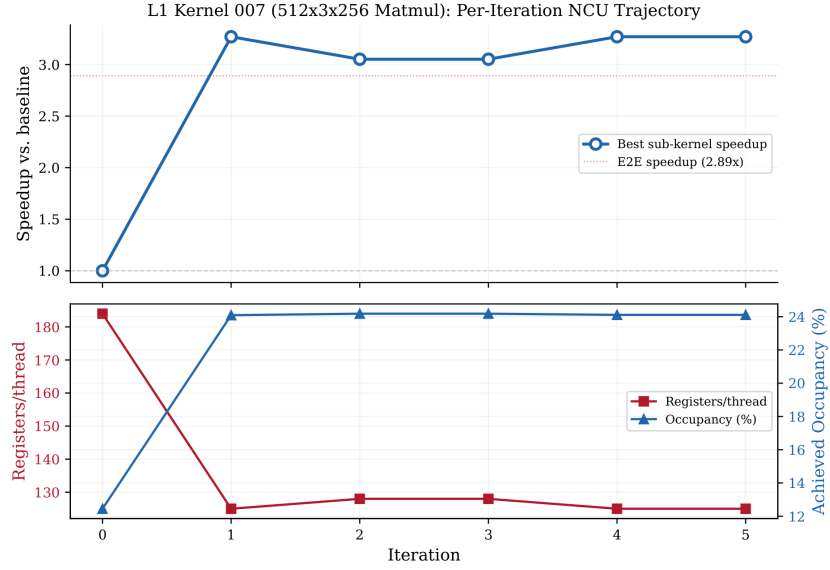


Figure 6: Optimization trajectory for L1-007 (matmul). Each point represents a candidate kernel evaluated during beam search. The planner identifies register pressure (184 regs, 12.4% occupancy) as the primary bottleneck from NCU metrics. Iteration 1, beam B applies FP16 dot-product accumulation, reducing registers to 125 and achieving a $3.27\times$ sub-kernel speedup ($2.89\times$ E2E after model-level overhead).

Component	Specification
GPU	NVIDIA H200 SXM (Hopper, CC 9.0)
SMs	132
HBM3	141 GB, 4.8 TB/s bandwidth
L2 Cache	50 MB
CUDA	12.8
PyTorch	2.7.0
Triton	3.3.0
NCU	2025.1.0
Python	3.10 / 3.11
LLM	Claude Sonnet 4.6 (Anthropic)
LLM API	Google Cloud Vertex AI
LLM timeout	900 s per call
Beam width	4 chains
Iterations	5 per kernel
Plans per iter	4 (one per chain)
Retries (K)	4 per plan
E2E retries	3
Perf gate γ	1.03 (3% margin)

Table 11: Hardware, software, and hyperparameter configuration used for all experiments.