From Skills to Tracelets: Dependency-Guided Transfer for LLM Kernel Agents

Shuoming Zhang, Ruiyuan Xu, Qiuchu Yu, Guangli Li, Huimin Cui, Jiacheng Zhao
SKLP, Institute of Computing Technology, Chinese Academy of Sciences
University of Chinese Academy of Sciences

***Abstract*—GPU optimisation knowledge is now externalised at scale—human cookbooks, agent-mined failure rules, expert-derived skill libraries—yet most LLM kernel agents still consume it as isolated prompt hints. We argue that the natural unit of transfer is neither a final kernel nor a bare skill, but an *optimisation tracelet*: a small dependency-aware fragment of an optimisation trajectory whose actions are intent-level edits anchored to structural positions (loops, buffers, workers, sync scopes). Tracelets are IR-agnostic—any anchorable surface (Triton, a compiler IR, or annotated source) admits them—and the goal is to make optimisation knowledge a *programmable* layer that composes across non-identical GPU kernels, spanning shape, operator, platform, and language. Compared with prompt-only reuse or template replay, the reusable layer is dependency structure with declared anchors; target-side anchors and parameters are agentically re-materialised under validation. We report the tracelet formulation and a validation-gated trace-prefix search, TRACELETTRANSFER, that adapts source tracelets to nearby targets by re-solving anchors, retuning knobs, and validating trace prefixes. In a preliminary nearby-transfer study, TRACELETTRANSFER reaches or exceeds SOTA references within a small search budget, while ablations show that both the tracelet representation and dependency-guided prefix search are necessary for stable transfer.**

I. INTRODUCTION

LLM-based agents are becoming a practical interface for GPU-kernel optimisation: they draft an implementation in CUDA, Triton, or TileLang [23], run it through a compile/test/profile harness, inspect the feedback, and iterate. Benchmarks such as KernelBench [18] and TritonBench [14] make this loop measurable, and a growing line of agent systems pushes it forward through iterative refinement [25], [34], [35], multi-agent or evolutionary search [11], [31], post-training on agent trajectories [2], [7], [16], and RL-driven Triton synthesis [15]; recent surveys map this emerging area [30]. As these systems mature, attention has shifted from optimising any single kernel in isolation to *reusing* optimisation experience across kernels. Optimisation knowledge is now externalised at scale and in heterogeneous forms: human-authored CUDA / Triton / TileLang cookbooks; failure rules mined from agent traces [8]; profile-state memories that learn which optimisations tend to pay off under which bottleneck signals [6]; expert-derived skill memories and task-local trajectory records [21]; and reasoning graphs over optimisation-state transitions searched by MCGS [10]. The remaining question is different: once an optimisation is chosen, what exactly is the transferable object that should be carried to a new kernel?

Intents, not free-form edits. A consistent finding across recent work is that letting an LLM rewrite expert CUDA *directly* is unreliable, while rewriting through *some* intermediate representation that exposes structural positions is dramatically more so. CUBRIDGE [17] reports this gap on attention adaptation; the same pattern recurs across schedule-tree compilers and tensor program languages [3], [9], [19], [33], structured DSLs such as Triton [23], and richer compiler infrastructures like MLIR [13]: optimisation edits should be expressed as *intents*—high-level optimisation actions (tile, buffer, pipeline, ...) anchored to structural positions (which loop, which buffer, which worker, which sync scope)—rather than free-form code mutations. A parallel point is being made in agent tooling more broadly: Claude Code's two-tier skill packages [1] treat optimisation knowledge as authored skills with structural cues, not as raw code edits. We adopt this lesson without committing to a particular IR—the anchor surface can be a Triton-level structural view, a compiler IR, or structured natural-language annotations on source code—and call this requirement IR-in-the-middle; what matters is anchorability, not which IR.

Only intents are not enough Position is necessary but not sufficient. GPU optimisation often has pass-like prerequisite requirements: an edit may be invalid or locally harmful until another edit has installed the right state. Consider **shared-memory tiling**, **double buffering**, and **software pipelining** as a canonical chain. Pipelining is invalid before staged data movement exists. Double buffering, in isolation, may introduce extra movement and synchronisation and slow the kernel down. Yet the three together enable overlap of memory traffic with compute and are routinely profitable. A very strong model might have internalised that these steps should be applied as a chain; an ordinary prompt-only or skill-guided search has to discover that ordering. Greedy optimisation prunes double buffering as locally unattractive, while broader search often reaches the full chain only late in the budget. We call this configuration a *prerequisite valley*: a locally unattractive step must be crossed before a downstream optimisation becomes useful. A method-level skill or profile-state recommendation can be evidence-gated, but by itself it does not say which target region must carry each edit, nor which other anchored edits must be preserved so that a locally weak step becomes

worthwhile.

Tracelets: intent-level edits with dependency. These observations point to the missing transfer unit. It must be higher-level than code, because replaying a source kernel fixes the least portable details. It must be position-aware, because an optimisation only makes sense at the loop, buffer, worker, or sync scope where its preconditions hold. And it must be history-aware, because the value of a step such as double buffering may only appear after a downstream step reuses the same structural state. We therefore propose *optimisation tracelets*: small dependency-aware fragments of optimisation trajectories. A tracelet records (i) a set of intent-level edits, each anchored to a structural position; (ii) dependency constraints among anchors (B presupposes A's anchor being installed); (iii) expected local effects, including delayed payoff for dependency-bridging steps; and (iv) adaptation knobs that may need re-solving on a new target. A tracelet thus transfers *dependency structure with declared anchors*—not code, not isolated skills.

Why agentic transfer. Transfer enters because the source and target kernels are rarely identical. Across nearby targets, the dependency structure of an optimisation chain is often the most stable part; the concrete anchors are only partially stable, and tile sizes, stage counts, vector widths, and predicates usually have to be re-solved. Existing agent memory systems contribute method selection, profile evidence, failure avoidance, and trajectory search—in the spirit of broader experiential agent frameworks [24], [32] but specialised to kernel optimisation—yet none carries target-region bindings or declared prerequisite edges as the reusable unit (Appendix B). A tracelet fills this gap by making *what*, *where*, and *after what* first-class. The LLM agent is therefore not citing a skill or copying a reference kernel; it acts as an *anchor resolver* and *local repair generator* that re-materialises the same dependency-bearing edit history on a new target under the harness's validation gate.

Scope. We do not study how tracelets are mined and we do not propose a new general-purpose search algorithm; conditioned skill extraction, lineage construction, and trace mining are complementary ways of *producing* reusable optimisation knowledge. We assume a source tracelet exists and ask how an LLM kernel agent should *consume* it when the target is nearby enough for dependency structure to matter but different enough that direct replay is brittle. The evaluation uses a controlled *axis-isolated nearby-transfer* protocol: case, platform, and surface-language variants each change one dominant facet at a time.

We make three contributions.

- **Optimisation tracelets.** We formulate transferable optimisation knowledge as intent-level edits with structural anchors and dependency constraints, IR-agnostic over any anchorable surface.
- **Dependency-guided transfer.** We present TRACELETTRANSFER, a validation-gated prefix search that preserves dependency structure while re-solving anchors, retuning knobs, and applying local repairs on a target kernel.
- **Nearby-transfer study.** We define an axis-isolated nearby-transfer protocol across case, platform, and surface-language changes, and compare flat skill reuse, unguided tracelet reuse, and full TRACELETTRANSFER under the same budget. The results show that both anchored tracelets and dependency-guided prefix search are useful for rapidly adapting source tracelets to nearby targets; a separate composition case study illustrates how multiple tracelets can be combined on a new target.

Full picture. The larger arc is a *programmable optimisation substrate* for GPU kernel agents: a tracelet library that composes across shape, operator, platform, and language; an anchor surface tightened into a structured IR with closed-loop verification; and an agentic re-materialisation loop that handles structurally distant targets—CUDA $\leftrightarrow$ Triton, Hopper $\leftrightarrow$ Ampere, GEMM mainloop $\rightarrow$ FlashAttention-style fused mainloop—with the same dependency-guided machinery. The current scope isolates the smallest piece of that substrate: what it means to transfer a dependency-bearing optimisation fragment when one is given.

II. METHOD: TRACELETTRANSFER

We show the overview of TRACELETTRANSFER in Figure 1. Given externally supplied source tracelet, TRACELETTRANSFER turns a naive kernel K_0 into a partially-optimised target kernel K_{mid} with tracelet-guided nearby transfer.

A. Tracelets and where they come from

A bare skill names an optimisation; a tracelet names how a small sequence of optimisations must be installed, we document the schema of tracelet in Appendix A. Each tracelet action says *what* intent to apply, *where* it is anchored, and which target-side knobs may change. Edges record prerequisite constraints among those anchored actions. The running example is the canonical chain:

$$A = \texttt{tile} \quad \rightarrow \quad B = \texttt{buffer} \quad \rightarrow \quad C = \texttt{pipeline}.$$

Here A installs shared-memory tiling on the mainloop K-loop; B adds double buffering at the asynchronous copy sites created by A; and C pipelines the same K-loop using B's buffers. The important fact is not merely that these three skill names co-occur, but that they must be anchored compatibly. Double buffering is the locally weak step in the prerequisite valley: it may slow the kernel before pipelining is installed, yet it is the state that makes pipelining valid and profitable.

Where tracelets come from. The formulation does not prescribe a single source. Tracelets can be *lifted* from an existing expert kernel via an IR-aware codegen pipeline that records its execution-orchestration history (CuBridge's CuIR is one attention-targeted instance [17], and hand-engineered kernels such as the FlashAttention series [4], [5], [20] are natural lifting sources); they can be *adapted from open skill or method libraries* [21] by attaching anchors and prerequisite

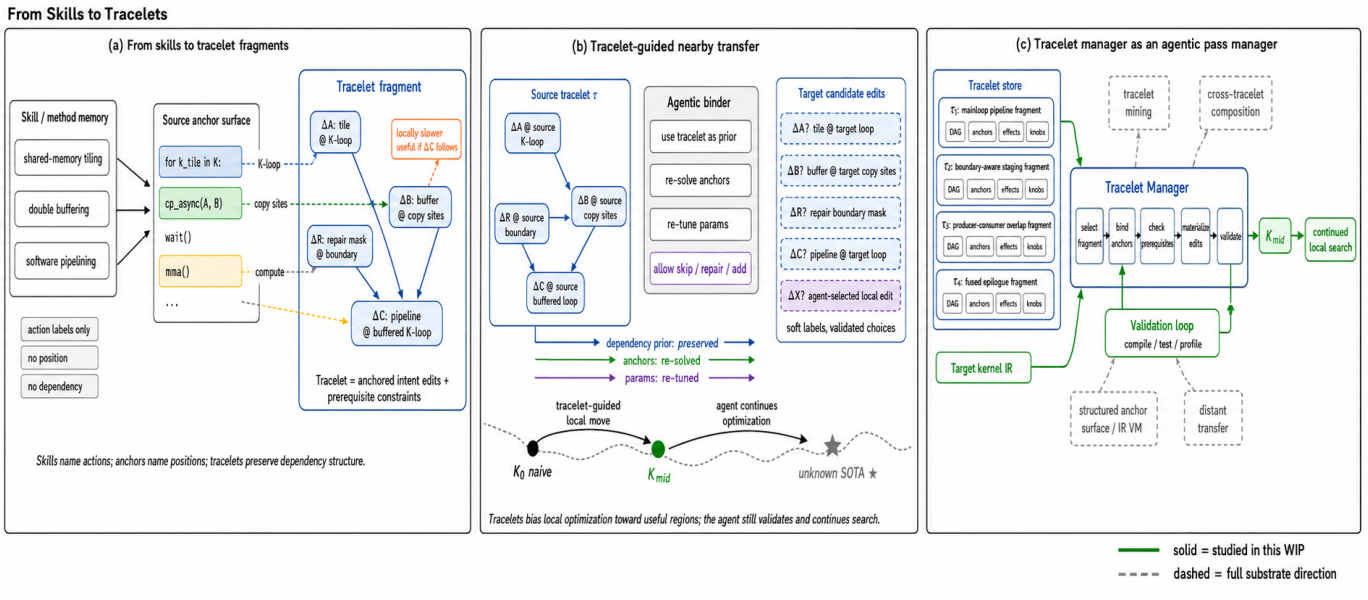

Fig. 1. **From skills to tracelets.** A tracelet is the transfer object between skill memories and concrete kernels: it lifts action labels into anchored intent edits, records prerequisite structure as a DAG rather than a hard script, and uses that structure as a soft prior when re-binding edits to a nearby target. The current WIP studies the solid path—an existing source tracelet, a nearby target kernel, an agentic binder, and compile/test/profile validation that produces K_{mid}. Dashed components indicate the broader agentic optimisation substrate, including external tracelet production and cross-tracelet composition; §IV sketches that direction.

edges to method recommendations that were originally label-level; and they can be *augmented dynamically* during transfer itself, when a local repair or extra prerequisite becomes a useful intent edit and is added back to the tracelet for downstream targets. The present scope assumes a source tracelet is given; §IV returns to the production question.

B. Tracelet transfer

TRACELETTRANSFER focuses on *axis-isolated nearby transfer*: each target changes at most one dominant facet of the source context while preserving enough structural roles for the tracelet to be re-bound. Examples include case transfer (a GEMM tracelet retargeted to conv2d via implicit-GEMM), platform transfer (an A100-derived sm80 tracelet → an sm120 RTX PRO 6000 target, an Ampere-class kernel rebound on a consumer-grade Blackwell-class GPU that adds TMA), and surface-language transfer (a CUDA-derived tracelet materialised on Triton or TileLang). These cases are nearby not because the source code is copyable, but because one layer of the optimisation history remains reusable.

The working hypothesis is that *dependency structure is the most stable layer* under such transfers. Anchors are partially stable and often re-projectable; parameters and effects are the least stable and usually need target-side search or validation. Replay fails because it fixes the least stable layers. Prompt-only reuse fails because it does not preserve the dependency roles strongly enough for the agent to cross a prerequisite valley.

Search node and merging. Given a source tracelet τ and a target kernel K_0, TRACELETTRANSFER explores trace prefixes on the target. A search node is a triple $(P, K, A_{\mathrm{state}})$: the installed prerequisite set, the current kernel, and the anchor choices made so far. To extend a node, the agent picks an enabled action whose prerequisites in E already hold, resolves its anchor on K via pattern or role match (Appendix A), and emits an intent edit. The harness materialises the edit and runs the compile/test/profile gate; valid prefixes that install the same prerequisite set under compatible anchors are merged ($A{\to}B$ and $B{\to}A$ collapse to $\{A, B\}$), in the spirit of equality saturation [22], [27], [29] but bounded by tracelet-level anchor compatibility rather than full program equivalence.

Output: K_{mid}. TRACELETTRANSFER returns the best validated adapted prefix as the starting kernel K_{mid}. The downstream LLM agent then continues ordinary local search from K_{mid}. The (K_{mid}, residual constraints) pair is search-policy-agnostic: tree search, evolutionary search, and sequential refinement all consume it identically. TRACELETTRANSFER does not propose a new search algorithm above K_{mid}—it changes *where the search starts* and *which actions can be proposed*. Algorithm 1 writes the loop in pseudocode. The three concrete edit shapes used during re-materialisation (knob retuning, anchor re-solution, local repair) are catalogued in Appendix C; the comparison with compiler autotuning is in Appendix D.

III. EVALUATION

A. Setup

We evaluate three axis-isolated nearby-transfer settings (Table I) and one separate composition case study. Each row fixes one target case at a single shape so that latencies are comparable within the row: a CNN-sized conv2d via implicit-GEMM, a square matmul-friendly GEMM on RTX PRO

TABLE I

PRELIMINARY RESULTS ON THREE NEARBY-TRANSFER SETTINGS AND ONE NEW-TARGET COMPOSITION CASE. THE FIRST THREE ROWS ISOLATE TRANSFER ALONG ONE DOMINANT FACET—CASE, PLATFORM, OR SURFACE LANGUAGE—WHILE THE MHC ROW STUDIES HOW TRACELETTRANSFER CONSTRUCTS A NEW TARGET BY RE-BINDING AND COMPOSING MULTIPLE TRACELETS UNDER VALIDATION. THE SOTA COLUMN REPORTS ABSOLUTE LATENCY IN μS; OTHER COLUMNS REPORT SPEEDUP RELATIVE TO SOTA. **SKILL-FLAT** USES AN UNORDERED SKILL LIST, **TRACELET, UNGUIDED** KEEPS ANCHORS AND DEPENDENCIES WITHOUT GUIDED PREFIX SEARCH, AND **OURS** IS FULL TRACELETTRANSFER. ALL VARIANTS USE THE SAME BACKBONE, HARNESS, AND 30-ROUND BUDGET.

Setting	Source	Target	Skill-flat	Tracelet, unguided	Ours	SOTA
			×	×	×	μs
case	GEMM	conv2d (sm90)	0.66×	0.79×	1.24×	19.6
platform	GEMM on A100 (sm80)	GEMM on RTX PRO 6000 (sm120)	0.31×	0.74×	0.96×	369.6
language	CUDA conv2d	Tilelang conv2d (sm90)	0.49×	0.98×	1.25×	19.5
new	mHc connection [28]		0.68×	0.85×	1.08×	6.4 [26]

Algorithm 1: Dependency-Guided Tracelet Transfer

Input: source tracelet $\tau = (V, E, \mathit{effects}, \mathit{knobs}, \mathit{scope})$; target kernel K_0; budget B.
Init: frontier $\leftarrow \{(\emptyset, K_0, \emptyset)\}$; best $\leftarrow K_0$.
while budget remains **do**
  pick a node (P, K, A_{state}) from frontier;
  choose enabled $v \in V$ (or local repair) whose prerequisites in E hold and whose anchor resolves on K given A_{state};
  resolve $v.\mathit{anchor}$ on K's structure;
  instantiate v with $v.\mathit{params}$ *(LLM emits intent edit; harness materialises it on K)*;
  compile/test/profile the resulting K';
  if K' valid **then**
    $P' \leftarrow P \cup \{v\}$; $A'_{\text{state}} \leftarrow A_{\text{state}} \cup \{v.\mathit{anchor} \mapsto \text{resolved}\}$;
    merge $(P', K', A'_{\text{state}})$ with existing nodes installing the same prerequisite set under compatible anchors;
    add to frontier; update best if K' improves latency;
  else record failure; optionally propose a local repair.
return $K_{\text{mid}} \leftarrow$ best validated prefix.

6000 (sm120), the same GEMM ported into Tilelang, and a new open-ended mHc kernel generation problem. The table reports three axis-isolated nearby-transfer settings and one new-target composition study. The first three rows isolate transfer along a single dominant axis—case, platform, or surface language. The case serves as a preliminary study of how TRACELETTRANSFER works on a new target by re-binding and composing multiple tracelets under validation. We use Claude Code version 2.1.87 as the agent harness and Claude-Opus-4.6 as the backbone LLM for all variants.

Two-layer ablation. The framework's claim is layered, and Table I tests both layers. *Skill-flat* exposes the same actions as a flat skill list, isolating the gain from organising knowledge as tracelets. *Tracelet, unguided* keeps anchors and dependency edges but explores prefixes without dependency-guided ordering, isolating the gain from guided search. *Ours* is the full system. All three variants share backbone, prompt, harness, and a 30-round compile/test/profile budget; they differ only in how optimisation knowledge is presented and searched.

B. Metrics

We report speedup relative to the SOTA reference (SOTA latency / variant latency) for each variant; the SOTA column reports the absolute SOTA latency in microseconds for scale. The prose additionally reports *rounds-to-best*, since the second-layer claim is about exploration efficiency rather than peak performance.

C. Results

Table I shows that both layers of TRACELETTRANSFER matter. Anchored tracelets outperform a flat skill list because they preserve where each optimisation intent should bind and which prerequisites must be installed: *Skill-flat* stalls at $0.31\times$–$0.66\times$ of SOTA across the three nearby-transfer rows, while *Tracelet, unguided* already reaches $0.74\times$–$0.98\times$ on the same rows. Adding dependency-guided prefix search closes the remaining gap to or above SOTA: TRACELETTRANSFER reaches $0.96\times$–$1.25\times$ within the 30-round budget, meeting cuDNN on conv2d, exceeding it on the Tilelang port, and landing within 4% of CUTLASS-tuned on the sm120 GEMM. The mHc composition row shows the same qualitative pattern when multiple tracelets must be re-bound and composed to construct a new target: $0.68\times \rightarrow 0.85\times \rightarrow 1.08\times$ across the three variants.

IV. DISCUSSION

The full picture sketched in §I requires two pieces this paper does not provide. First, a tracelet library worthy of the name has to be *produced* rather than authored—lifted from expert kernels [17], [20] or mined from agent trajectories [6], [8], [21]—and independently produced tracelets must be checked for anchor compatibility before they compose (e.g. a CUTLASS pipelining tracelet plus a FlashAttention-3 fused-epilogue tracelet). Second, the anchor surface should tighten from prose into a structured representation (TVM IR / TensorIR [9], [33], or CuIR-style execution-orchestration IR [17]) so that validation moves from compile/test/profile to IR-level checks. These are the dashed components in Figure 1.

REFERENCES

[1] Anthropic, "Skills — Claude Docs," https://docs.claude.com/en/docs/claude-code/skills, 2025, official Claude Code documentation.

[2] C. Baronio, P. Marsella, B. Pan, S. Guo, and S. Alberti, "Kevin: Multi-turn rl for generating cuda kernels," *arXiv preprint arXiv:2507.11948*, 2025.

[3] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to optimize tensor programs," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[4] T. Dao, "Flashattention-2: Faster attention with better parallelism and work partitioning," in *International Conference on Learning Representations*, vol. 2024, 2024, pp. 35 549–35 562.

[5] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," *Advances in neural information processing systems*, vol. 35, pp. 16 344–16 359, 2022.

[6] K. S. Dong, S. Modi, D. Nikiforov, S. Damani, E. Lin, S. K. S. Hari, and C. Kozyrakis, "Kernelblaster: Continual cross-task cuda optimization via memory-augmented in-context reinforcement learning," *arXiv preprint arXiv:2602.14293*, 2026.

[7] H. Du, Q. Ge, J. Hu, A. Yang, Z. Cai, Z. Huang, S. Yuan, Q. Cheng, X. Xie, Y. Chen *et al.*, "Kernel-smith: A unified recipe for evolutionary kernel optimization," *arXiv preprint arXiv:2603.28342*, 2026.

[8] W. Du, J. Zhuo, Y. Dong, A. W. He, W. Sun, Z. Zheng, M. Karunaratne, I. Fox, T. Dettmers, T. Chen *et al.*, "Adaexplore: Failure-driven adaptation and diversity-preserving search for efficient kernel generation," *arXiv preprint arXiv:2604.16625*, 2026.

[9] S. Feng, B. Hou, H. Jin, W. Lin, J. Shao, R. Lai, Z. Ye, L. Zheng, C. H. Yu, Y. Yu *et al.*, "Tensorir: An abstraction for automatic tensorized program optimization," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 804–817.

[10] J. Gong, Z. Wei, J. Chen, C. Liu, and H. Li, "From large to small: Transferring CUDA optimization expertise via reasoning graph," in *The Fourteenth International Conference on Learning Representations*, 2026. [Online]. Available: https://openreview.net/forum?id=vqESUhcSOG

[11] R. T. Lange, Q. Sun, A. Prasad, M. Faldor, Y. Tang, and D. Ha, "Towards robust agentic cuda kernel benchmarking, verification, and optimization," *arXiv preprint arXiv:2509.14279*, 2025.

[12] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[13] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 2–14.

[14] J. Li, S. Li, Z. Gao, Q. Shi, Y. Li, Z. Wang, J. Huang, W. WangHaojie, J. Wang, X. Han *et al.*, "Tritonbench: Benchmarking large language model capabilities for generating triton operators," in *Findings of the Association for Computational Linguistics: ACL 2025*, 2025, pp. 23 053–23 066.

[15] S. Li, Z. Wang, Y. He, Y. Li, Q. Shi, J. Li, Y. Hu, W. Che, X. Han, Z. Liu *et al.*, "Autotriton: Automatic triton programming with reinforcement learning in llms," *arXiv preprint arXiv:2507.05687*, 2025.

[16] X. Li, X. Sun, A. Wang, J. Li, and C. Shum, "CUDA-l1: Improving CUDA optimization via contrastive reinforcement learning," in *The Fourteenth International Conference on Learning Representations*, 2026. [Online]. Available: https://openreview.net/forum?id=igZItUbY6n

[17] X. Ma, Y. Zhou, W. Sun, Z. Liu, J. Leng, Y. Lin, S. Sun, M. Guo, and J. S. Dong, "Cubridge: An llm-based framework for understanding and reconstructing high-performance attention kernels," *arXiv preprint arXiv:2605.05023*, 2026.

[18] A. Ouyang, S. Guo, S. Arora, A. L. Zhang, W. Hu, C. Re, and A. Mirhoseini, "Kernelbench: Can llms write efficient gpu kernels?" in *International Conference on Machine Learning*. PMLR, 2025, pp. 47 356–47 415.

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

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

[21] Q. Sun, J. Han, T. Li, Z. Tang, S. Chen, F. Yang, A. Liu, X. Liu, and Y. Liu, "Kernelskill: A multi-agent framework for gpu kernel optimization," *arXiv preprint arXiv:2603.10085*, 2026.

[22] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality saturation: a new approach to optimization," in *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2009, pp. 264–276.

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

[24] G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan, and A. Anandkumar, "Voyager: An open-ended embodied agent with large language models," *Transactions on Machine Learning Research*, 2024. [Online]. Available: https://openreview.net/forum?id=ehfRiF0R3a

[25] J. Wang, V. Joshi, S. Majumder, X. Chao, B. Ding, Z. Liu, P. P. Brahma, D. Li, Z. Liu, and E. Barsoum, "Geak: Introducing triton kernel ai agent & evaluation benchmarks," *arXiv preprint arXiv:2507.23194*, 2025.

[26] X. Wang, C. Xu, H. Cao, R. Tian, W. Zhao, K. Yu, and C. Zhao, "Tilekernels," https://github.com/deepseek-ai/TileKernels, 2026.

[27] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "Egg: Fast and extensible equality saturation," *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, pp. 1–29, 2021.

[28] Z. Xie, Y. Wei, H. Cao, C. Zhao, C. Deng, J. Li, D. Dai, H. Gao, J. Chang, K. Yu *et al.*, "mhc: Manifold-constrained hyper-connections," *arXiv preprint arXiv:2512.24880*, 2025.

[29] Y. Yang, P. Phothilimthana, Y. Wang, M. Willsey, S. Roy, and J. Pienaar, "Equality saturation for tensor graph superoptimization," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 255–268, 2021.

[30] Y. Yu, P. Zang, C. H. Tsai, H. Wu, Y. Shen, J. Zhang, H. Wang, Z. Xiao, J. Shi, Y. Luo *et al.*, "Towards automated kernel generation in the era of llms," *arXiv preprint arXiv:2601.15727*, 2026.

[31] Z. Zhang, R. Wang, S. Li, Y. Luo, M. Hong, and C. Ding, "Cudaforge: An agent framework with hardware feedback for cuda kernel optimization," *arXiv preprint arXiv:2511.01884*, 2025.

[32] A. Zhao, D. Huang, Q. Xu, M. Lin, Y.-J. Liu, and G. Huang, "Expel: Llm agents are experiential learners," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, no. 17, 2024, pp. 19 632–19 642.

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

[34] Q. Zhou, S. Peng, W. Xiong, H. Chen, Y. Wen, H. Li, L. Li, Q. Guo, Y. Zhao, K. Gao *et al.*, "Qimeng-attention: Sota attention operator is generated by sota attention algorithm," in *Findings of the Association for Computational Linguistics: ACL 2025*, 2025, pp. 8491–8505.

[35] X. Zhu, S. Peng, J. Guo, Y. Chen, Q. Guo, Y. Wen, H. Qin, R. Chen, Q. Zhou, K. Gao *et al.*, "Qimeng-kernel: Macro-thinking micro-coding paradigm for llm-based high-performance gpu kernel generation," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 40, no. 34, 2026, pp. 29 168–29 176.

APPENDIX A
TRACELET SCHEMA AND ANCHOR SURFACE

A tracelet is written compactly as

$$\tau = (V, E, \mathit{effects}, \mathit{knobs}, \mathit{scope})$$

with each action

$$v = (\mathit{action\_kind}, \mathit{anchor}, \mathit{params}).$$

action_kind names an intent such as `tile`, `buffer`, `pipeline`, `mask_split`, or `vectorize`. *params* are the target-side knobs: tile sizes, buffer depth, stage count, vector width, or boundary predicates. Edges in E are prerequisite constraints over anchors; for example, $B \rightarrow C$ says that C's pipeline anchor presupposes the buffers installed by B at a compatible position.

Scope as a soft (C, L, P, D) tag. *scope* is a four-axis soft tag recording the source context the tracelet was authored under: C is the algorithmic case (e.g. "GEMM mainloop", "softmax-with-causal-mask"); L is the surface language (CUDA, Triton, TileLang); P is the platform (sm80 / sm90 / sm120, ...); and D summarises data shape and type relevant to the optimisation (e.g. $M{=}N{=}K$ multiples of 128, fp16). The tag is *soft* on purpose: it is a hint about where the tracelet was validated, not a hard precondition. The four axes correspond directly to the four facets of axis-isolated nearby transfer in §III: a transfer task asks for re-binding to a target whose tag differs from *scope* along one axis. Hard scope predicates (in the spirit of [8]) are not mandatory at this layer; if a tracelet author wishes to assert one, it can be encoded as a guard inside an action's preconditions.

Anchor as an IR-intent fragment. An *anchor* is written as a small declarative fragment over the chosen anchor surface's named primitives, not as a free-form sentence. Two equivalent shapes are admissible:

- *Pattern fragment*: a structural pattern over the surface, e.g. `loop[role=reduction] over operand[stage=smem]`, which matches any loop that already reduces over a shared-memory-staged operand.
- *Role fragment*: a reference to another action's installed state, e.g. A`.K-loop` or B`.cp_async_sites`, which delegates the structural match to a previously resolved action.

Pattern fragments are first-class re-bindable; role fragments are local references that the prefix-search resolves transitively. We omit a third *pinned* form (e.g. "line 42" or "loop_id_42") because it amounts to replay and defeats transfer. A pattern fragment plus a role fragment together suffice for the cross-kernel cases evaluated in §III; nothing in the formulation prevents a richer anchor language if the surface supports it.

Anchor-surface contract. The formulation does not require one particular IR, but it does require an *anchor-surface contract*. The surface must (i) expose named structural regions reachable by pattern or role fragments; (ii) support resolving a tracelet anchor against a target kernel; (iii) allow an intent edit to be materialised at the resolved region; and (iv) provide a validation gate for correctness and performance feedback. A Triton AST, TensorIR schedule, CuIR-style execution- orchestration IR, or structured source annotation can all serve this role if they satisfy the contract.

Worked example: the running $A{\to}B{\to}C$ tracelet. The tracelet of §II-A can be written as

```
tracelet pipeline_chain {
  scope: (C="GEMM mainloop", L="CUDA",
          P="sm80", D="M,N,K%128=0; fp16")

  A: tile @
       loop[role=reduction]
         over operand[layout=row|col]
       params { BM, BN, BK }
       effect: small gain; enables A.K-loop

  B: buffer @ A.cp_async_sites
       params { depth }
       effect: slight slowdown; enables B.buffers

  C: pipeline @ A.K-loop with B.buffers
       params { stages }
       effect: large gain (overlap)

  edges: A -> B, B -> C, A -> C
}
```

At authoring time only A's anchor is a pattern fragment; B and C's anchors are role fragments that compose with A's resolved position. Re-binding to a new target therefore reduces to resolving A's pattern fragment on the target surface and propagating roles through the dependency edges, with knobs retuned and validation gating each materialised prefix.

APPENDIX B
POSITIONING RELATIVE TO AGENT MEMORY SYSTEMS

A growing body of agent memory and skill systems for kernel optimisation makes the agent's choice of *what to try next* less improvised. The point of this appendix is not to diminish them. They solve complementary problems—method selection, evidence gating, failure avoidance, trajectory search—and tracelets are designed to sit on top of, rather than replace, what they produce. The tracelet claim is narrower: when an optimisation is to be carried to a new kernel, the reusable object should also carry target-region bindings and prerequisite relations as first-class structure, not as implicit detail of an example or as label-level co-occurrence in a graph. Below we walk through six representative reusable objects, ending with our own.

A reference or few-shot kernel. The simplest reusable object is a complete expert kernel, used either as a reference implementation or as an in-context example. Such an object is invaluable for understanding *what good looks like*, but the structural roles in it—which loop is the K-loop, which copies are staged, which warps are producers—live entirely in the source code, and the target binding is left to be inferred by whatever consumes the example. A reference kernel transfers the destination, not the dynamics of getting there.

AdaExplore-style failure memory. AdaExplore-style memories [8] record failure rules and adaptation traces, mined from agent attempts and used to guide search or trigger repair. They make a key contribution: they turn dead-end attempts into structured negative knowledge. The reusable object is a *condition under which a step fails*, not a re-bindable edit; an AdaExplore memory tells the next agent "avoid this kind of move here" without committing the surviving moves to a transferable prerequisite chain.

KernelBlaster-style memory. KernelBlaster [6] learns persistent profile-state $\to$ optimisation/score memories using in-context RL, conditioning the next move on bottleneck signals at the current state. The mechanism is genuinely state-conditioned: useful preparatory transitions can emerge as patterns in the memory. But the memory is keyed by profile signals and optimisation scores rather than by target-region

bindings, so a useful transition learnt on one kernel does not, by construction, name where on a new kernel it should be re-installed.

KernelSkill-style memory. KernelSkill-style memories [21] curate expert-derived method knowledge with evidence predicates, implementation cues, and task-local trajectory records. They are the closest existing object to a *skill*: each entry asserts "this method is appropriate when these conditions hold" with retrieval over the conditions. Implementation cues can mention code structure, but the long-term reusable unit remains a method recommendation rather than an anchored, dependency-bearing edit, and the trajectory memory mainly stabilises behaviour within the current task.

ReGraphT-style reasoning graph. ReGraphT [10] aggregates LLM-generated optimisation trajectories into a directed graph of method labels and uses Monte Carlo Graph Search to navigate it. The graph is an excellent search scaffold: it makes phase-order knowledge explicit at the *label* level (which methods follow which). What it does not carry, by design, is a target-region binding or an anchor-compatible prerequisite edge; the edges are statistical co-occurrences enriched with examples, useful for retrieval-augmented generation but not intended to be re-materialised on a structurally different kernel.

IR snapshot or operator-specific bridge. CuBridge [17] and similar systems demonstrate that lifting a kernel into a structured IR makes intent-level edits much more reliable for an LLM. Within the chosen IR, anchors are first class. However, the reusable object is a snapshot or a same-kernel adaptation: the prerequisite *history* that made an optimisation profitable is not what is carried, and the IRs are often engineered around one operator family.

Tracelets (ours). A tracelet is an intent-edit DAG with anchors, prerequisite edges, effects, knobs, and scope. The reusable object is the dependency structure with declared anchors: *what* to apply, *where* to anchor it on the target, and *after what* to apply it. Anchors are written as pattern or role fragments (Appendix A) and re-solved by the agent on the target kernel under validation. This is what existing memories are not designed to provide: a re-bindable optimisation history that survives non-identical targets.

Table II marks the same comparison along two axes that matter for transfer: whether the reusable object carries a target-region binding, and whether it carries an anchor-compatible prerequisite chain.

APPENDIX C
ADAPTATION TAXONOMY

The intent edit emitted by the agent at each search step falls into one of three concrete shapes, depending on how the target diverges from the source.

Knob retuning. The structural anchor is unchanged but $v.params$ must be retuned for the target: tile sizes, buffer depth, stage count, vector width, predicates. This is the dominant adaptation under shape and within-platform variants—e.g. a GEMM tracelet retuning $\{B_M, B_N, B_K\}$ when transferred from one shape to another.

TABLE II
TWO AXES THAT MATTER FOR CROSS-KERNEL TRANSFER: TARGET-REGION BINDING AND ANCHOR-COMPATIBLE PREREQUISITE CHAIN. EXISTING MEMORIES COVER OTHER AXES WELL (FAILURE RULES, PROFILE STATES, EVIDENCE PREDICATES, SEARCH SCAFFOLDS); NONE PROVIDE BOTH OF THE TWO TRANSFER-TIME AXES AS FIRST-CLASS STRUCTURE.

System / abstraction	Target-region binding	Prerequisite chain
Reference / few-shot kernel	implicit	implicit
AdaExplore failure memory [8]	×	×
KernelBlaster memory [6]	×	statistical
KernelSkill memory [21]	×	×
ReGraphT graph [10]	×	label-level
IR snapshot / bridge [17]	in-IR	snapshot
Tracelet (ours)	✓	✓

Anchor re-solution. Source and target differ structurally and $v.anchor$ must be re-projected onto the target's structure. Dependency edges remain unchanged but they now constrain the re-projected anchors. The canonical case is a GEMM-mainloop tracelet applied to implicit-GEMM Conv2D, where every anchor that pointed at "the K-loop" is re-projected onto the target's reduction loop over $(C_{\text{in}} \times K_H \times K_W)$. Anchor re-solution dominates cross-operator transfer.

Local repair. The agent inserts an extra anchored edit when a candidate prefix fails validation. Common shapes are adding a boundary predicate, splitting a loop into full and partial parts, or widening a vector load when the target's alignment differs. Each repair is itself an intent edit with its own anchor and a light dependency on existing anchors, and may be merged back into the tracelet for downstream targets—one of the three tracelet provenance routes in §II.

APPENDIX D
COMPILER-INFRASTRUCTURE ANALOGY

Why agentic, not just autotuned. Compiler autotuning [3], [9], [19], [33] already exemplifies a related asymmetry to a static *PassManager* [12], [13]: instead of varying the program under a fixed pipeline, autotuners fix one program and search over its schedule or optimisation sequence. The prerequisite valley TRACELETTRANSFER crosses is, viewed through that lens, a phase-ordering problem inherited from compilers, and prefix merging is a tracelet-level relative of equality saturation. What TRACELETTRANSFER adds is the *cross-program* axis that neither path covers natively: a previously useful optimisation sequence is carried to a new target whose structural positions have shifted, and the agent re-projects its anchors and retunes its knobs under validation rather than searching the whole sequence from scratch. The LLM is the anchor resolver and local-repair generator that closes this gap, and is what makes tracelets a *transferable* unit rather than a per-program tuning artefact.

What we inherit, and what we do not claim. A tracelet is, structurally, a small pass-pipeline fragment, and the prerequisite valley is the phase-ordering problem in disguise;

Axis	Compiler infra. (PassManager / autotuner)	Tracelet substrate (ours)
What varies	program / schedule (intra-program)	pipeline across non-identical programs
Reusable unit	pass / autotuned schedule	anchored intent-edit DAG with prereq. edges
Cross-program transfer	×	✓(anchor re-solution)
Validation	IR verifier; cost model	compile / test / profile + anchored repairs

TABLE III
COMPILER ANALOGY IN SKELETON FORM. STATIC INFRASTRUCTURE IS INTRA-PROGRAM; TRACELETS OCCUPY THE CROSS-PROGRAM AXIS WITH ANCHOR RE-SOLUTION AS THE AGENT'S JOB.

neither mechanism is a new claim on our part. We do not claim a verifier-style guarantee, an SSA-strength validator, or a cost model competitive with autotuner cost models: validation in TRACELETTRANSFER is the harness's compile/test/profile gate, cost is real-world latency, and prefix merging is bounded by tracelet-level compatibility on installed prerequisite sets, not by full kernel semantic equivalence. The structural analogy is what we use; we do not import the formal apparatus.

Table III marks the analogy as a 4-row contrast.