Towards Automated Kernel Generation in the Era of LLMs

Yang Yu[1], **Peiyu Zang**[1,2], **Chi Hsu Tsai**[1,3], **Haiming Wu**[1,4], **Yixin Shen**[1,5], **Jialing Zhang**[1,6], **Haoyu Wang**[1,7], **Zhiyou Xiao**[1,3], **Jingze Shi**[8], **Yuyu Luo**[8], **Wentao Zhang**[3], **Chunlei Men**[1], **Guang Liu**[1] and **Yonghua Lin**[1]
[1]Beijing Academy of Artificial Intelligence
[2]Beijing Normal University
[3]Peking University
[4]Beijing Institute of Technology
[5]Cornell University
[6]Beijing Jiaotong University
[7]Renmin University of China
[8]Hong Kong University of Science and Technology (Guangzhou)

Abstract

The performance of modern AI systems is fundamentally constrained by the quality of their underlying GPU kernels, which translate high-level algorithmic semantics into low-level hardware operations. Achieving near-optimal kernels requires expert-level understanding of hardware architectures and programming models, making kernel engineering a critical but notoriously time-consuming and non-scalable process. Recent advances in large language models and LLM-based agents have opened new possibilities for automating kernel generation and optimization. LLMs are well-suited to compress expert-level kernel knowledge that is difficult to formalize, while agentic systems further enable scalable optimization by casting kernel development as an iterative, feedback-driven loop. Rapid progress has been made in this area. However, the field remains fragmented and lacks a systematic perspective for LLM-driven kernel generation. This survey addresses this gap by providing a structured overview of existing approaches, spanning LLM-based approaches and agentic optimization workflows, and systematically organizing the datasets and benchmarks that underpin learning and evaluation in this domain. Moreover, key open challenges and future research directions are further outlined, aiming to establish a comprehensive reference for the next generation of automated kernel optimization. To keep track of this field, we maintain an open-source GitHub repository at https://github.com/flagos-ai/awesome-LLM-driven-kernel-generation.

1 Introduction

Rapid scaling of large language models (LLMs) has made efficient hardware utilization a central challenge in modern AI systems [Kaplan *et al.*, 2020]. Consequently, specialized accelerators such as GPUs and NPUs have become the foundation of large-scale training and inference [Choquette *et al.*, 2021; Liao *et al.*, 2021]. Their performance is largely governed by kernels implementing fundamental operations, such as matrix multiplication and attention, which dominate execution time in LLM workloads. As a result, overall system throughput, efficiency, and cost are often determined more by kernel quality than by hardware peak performance.

Despite their foundational role, the development of efficient kernels remains a formidable engineering challenge. Achieving near-peak hardware utilization requires deep expertise in both algorithmic design and hardware-specific intricacies. Furthermore, kernel optimization is inherently non-scalable: implementations are often tightly coupled to particular hardware architectures and workload characteristics, which hinders their reuse and generalization across different GPU generations or hardware vendors [Wu, 2023]. Although compiler-based autotuning approaches partially alleviate these scalability challenges through automated schedule generation and optimization, they remain fundamentally constrained by manually designed search spaces, scheduling primitives, and optimization priors.

In response to these challenges, LLMs and agentic systems have emerged as a promising paradigm for kernel generation and optimization. Trained on large-scale code repositories and technical documentation, LLMs encode substantial expertise in hardware-aware programming, enabling them to bridge the gap between high-level algorithmic specifications and low-level implementations. Beyond one-shot code generation, agentic frameworks leverage iterative execution feedback to refine candidate kernels, facilitating scalable and more open-ended exploration of complex optimization spaces across workloads and hardware platforms. Consequently, LLMs and agents are increasingly becoming a foundation for automated kernel development.

Despite rapid progress, research on LLM-driven kernel generation remains fragmented and lacks a systematic syn-

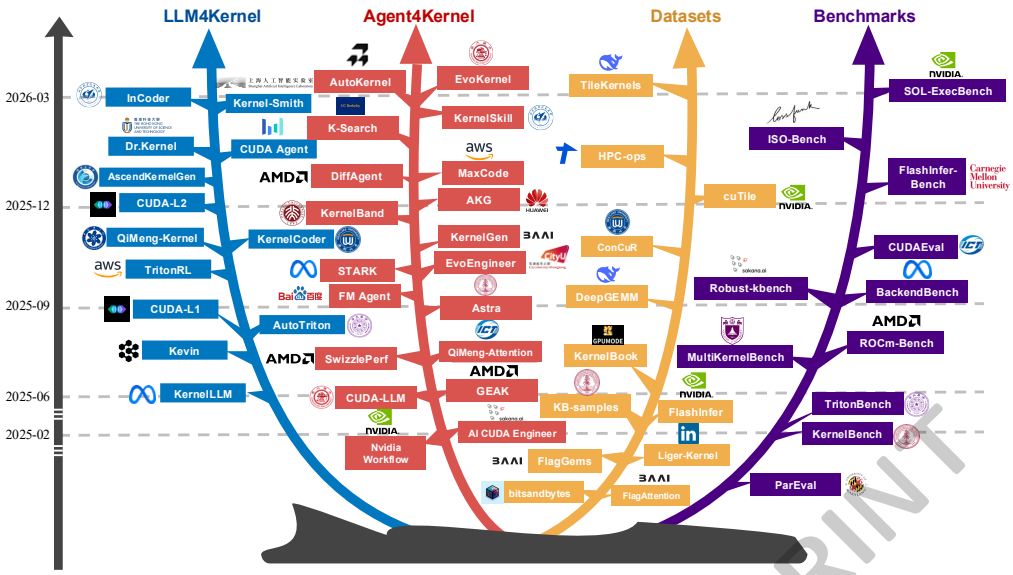

Figure 1: Illustration of the growth trend in the field of LLM-driven kernel generation. We organize these research works chronologically and categorically based on their publication dates and the domains they belong to.

thesis. This survey addresses this gap by presenting a unified overview of the field, clarifying foundational concepts, and highlighting emerging methodologies and trends. A key contribution is our consolidated resource infrastructure, which features a structured organization of training-ready kernel datasets and a literature collection tailored for retrieval-augmented generation (RAG), designed to facilitate data-driven research in this specialized kernel-generation domain. Moving beyond a synthesis of existing methodologies, we also highlight critical open challenges and propose promising research directions, aiming to establish a foundational reference for the next generation of innovation in LLM-driven kernel generation.

2 Background

LLMs and LLM-based Autonomous Agents. The foundation of modern LLMs is the Transformer architecture [Vaswani *et al.*, 2017], which functions as the probabilistic predictor trained via the next token prediction objective. Given a sequence of tokens $x = (x_1, \ldots, x_T)$, the model maximizes the joint probability:

$$P(x) = \prod_{t=1}^{T} P(x_t \mid x_1, \ldots, x_{t-1}; \theta).$$

This objective enables the model to internalize world knowledge and reasoning patterns implicitly during pretraining.

While LLMs serve as the cognitive engine for reasoning and decision-making, autonomous agents extend this capability by integrating additional system components such as planning, memory, and tool-use mechanisms, and interact with the environment through trial and error [Wang *et al.*, 2024]. In this framework, the LLM functions as the "brain", orchestrating actions through reasoning strategies. And agents utilize tools to perform actions beyond the model's internal knowledge and receive environmental feedback.

Kernel Programming and Code Generation. Kernel generation and optimization have traditionally followed two paradigms. The first relies on expert-written kernels and domain-specific abstractions, such as CUDA, CUTLASS, and TileLang, which enable fine-grained hardware optimization but require substantial architecture-specific expertise. The second centers on compiler-driven frameworks, such as Halide and TVM, which optimize kernels through scheduling and autotuning. While improving programmability, these approaches remain constrained by predefined search spaces, scheduling primitives, and handcrafted optimization rules. In contrast, LLM-based agents leverage knowledge distilled from large-scale code corpora and execution feedback to enable scalable and open-ended kernel optimization.

In parallel, LLMs have advanced code generation from code completion to complex software engineering tasks. However, kernel generation differs fundamentally from general-purpose code synthesis: beyond functional correctness, it must satisfy stringent performance requirements and adapt to hardware-specific execution characteristics. Consequently, it is more closely aligned with performance-oriented program synthesis and compiler optimization, necessitating specialized techniques beyond generic LLM-based code generation. This trend is also emerging in broader parallel programming systems, where LLMs have been applied to performance-oriented optimization beyond kernel generation [Wei *et al.*, 2025a].

3 LLM for Kernels Generation

Building on advances in LLM-driven code generation, recent work has increasingly applied LLMs to the generation

of high-performance GPU kernels. To highlight the methodological patterns that have emerged across this landscape, the following sections review two dominant families of post-training techniques used to specialize LLMs for kernel generation: *supervised fine-tuning* and *reinforcement learning*.

3.1 Supervised Fine-Tuning

Supervised fine-tuning (SFT) has become a central methodology for enabling LLMs to synthesize high-quality kernels, relying on paired datasets that capture both high-level computational intent and low-level kernel implementation patterns. KernelLLM [Fisches *et al.*, 2025] adopts this strategy by collecting samples and using the Triton compiler to produce aligned PyTorch–Triton examples, and applies instruction tuning with structured prompts that explicitly encode the mapping between computation and kernel structure. And ConCuR [Kong *et al.*, 2025] further generates and curates high-quality kernel datasets with reasoning traces, motivated by the observation that the structure and clarity of model reasoning can strongly affect kernel correctness and performance. Fine-tuning on such data leads to KernelCoder, a model capable of generating CUDA kernels that achieve 17% on the fast_1 metric of KernelBench [Ouyang *et al.*, 2025] level 1. Moreover, to address the stringent demands of industrial software development, InCoder-32B [Yang *et al.*, 2026] introduces a three-stage data curation pipeline consisting of pre-training, mid-training, and post-training, achieving a fast_1 score of 22.2% on KernelBench Level 1.

3.2 Reinforcement Learning

Reinforcement learning further enhances kernel generation via iterative feedback. Kevin [Baronio *et al.*, 2025] models kernel generation as a multi-turn optimization using cross-turn reward attribution for long-horizon credit assignment. CUDA-L1 introduces contrastive RL with an LLM-as-a-judge for dense feedback, and is refined by CUDA-L2 [Su *et al.*, 2025], which reports performance improvements over cuBLAS on its evaluated workloads. SparseRL [Wang *et al.*, 2026] leverages RL to generate high-performance CUDA code for sparse matrix operations. CUDA Agent [Dai *et al.*, 2026] introduces a large-scale agentic reinforcement learning system, including a skill-augmented CUDA development environment with automated verification and profiling to provide reliable reward signals, achieving state-of-the-art results on KernelBench, delivering a 99% faster rate over PyTorch Eager on KernelBench Level-1.

In addition to CUDA kernel generation, recent studies have investigated RL-based approaches for Triton kernel generation. AutoTriton [Li *et al.*, 2025d] generates Triton kernel with the model trained by RL and addresses reward sparsity by combining structural assessments of generated kernels with execution-based runtime rewards, while TritonRL [Woo *et al.*, 2025] extends this line of work through hierarchical reward decomposition and explicit verification of code outputs and intermediate reasoning traces. QiMeng-Kernel [Zhu *et al.*, 2025] further structures optimization by applying RL hierarchically to macro-thinking strategies rather than low-level implementation. Dr. Kernel [Liu *et al.*, 2026] introduces a robust distributed GPU environment for Triton kernel generations, combined with effective multi-turn RL methods to address the biased policy gradient issue and alleviates lazy optimization problem. Kernel-Smith [Du *et al.*, 2026] presents a stable evolution-oriented post-training recipe, and the Triton kernels generated by this framework achieve a 70% fast_1 score on KernelBench level 1. Finally, AscendKernelGen [Cao *et al.*, 2026b] expands the alignment learning paradigm to Ascend NPUs and generates AscendC kernels, combining CoT-based SFT with DPO (Direct Preference Optimization).

4 LLM Agent for Kernels Generation

Relying on foundational LLMs alone typically reduces kernel development to a static one-pass inference process. In contrast, LLM-based agents introduce autonomy and feedback into the optimization loop by enabling planning, tool use, and evaluation of intermediate results. This closed-loop, self-improving paradigm allows agent-based approaches to scale kernel optimization across diverse workloads and hardware platforms, while sustaining long-horizon, fatigue-free exploration. Concretely, we categorized recent agent-driven advancements into four structural dimensions: *learning mechanisms*, *external memory management*, *hardware profiling integration*, and *multi-agent orchestration*.

4.1 Learning Mechanisms

This category studies how LLM agents interact with execution environments and learn effective kernel generation strategies through trial-and-error. Initial approaches view kernel generation as iterative refinement. Caesar in KernelBench utilizes simple feedback loops to refine kernels, while Inference-Time Scaling [Chen *et al.*, 2025b] demonstrates that scaling test-time compute and reflection significantly boost kernel quality. To manage complexity, PEAK [Tariq *et al.*, 2025] employs a modular iterative stepwise refinement strategy. AutoKernel [Jaber and Jaber, 2026] profiles the full model to identify computational bottlenecks and employs an iterative optimization process to refine the corresponding kernel implementations. MaxCode [Ou *et al.*, 2026] further unifies existing iterative search methods under a max-reward reinforcement learning framework, combined with a natural language critique model converting raw execution feedback into diagnostic insights. K-Search [Cao *et al.*, 2026a] leverages LLMs as world models, utilizing their prior domain knowledge to guide the search process. It decouples high-level algorithmic planning from low-level program instantiation, and demonstrates strong performance on diverse and complex kernels.

Furthermore, a large body of agent-based methods employs iterative refinement for GPU kernel generation targeting diverse model architectures, programming languages, and hardware backends. For example, DiffAgent [Zhu *et al.*, 2026] adopts iterative refinement to accelerate diffusion models, TritonX [Hammond *et al.*, 2025] uses iterative refinement within a state machine to cover kernels of complete PyTorch ATen backends, and KernelGen [BAAI, 2025] leverages test-time scaling and reflection techniques to enable kernel generation for multi-chip backends. Moreover, many existing CLI-based agents such as Claude Code or OpenCode also

support iterative refinement, and AKO [Xie *et al.*, 2026] provides harnesses for such coding agents to enable agentic kernel optimization and achieve state-of-the-art performance on challenging benchmarks.

To escape local optima, recent frameworks adopt population-based evolution. Lange et al. [Lange *et al.*, 2025] optimize translation CUDA via mutation and crossover. FM Agent [Li *et al.*, 2025a] includes an evolutionary stage with the principles of diversity preservation, adaptive evolution, and multi-population dynamics. Advanced population dynamics are also introduced in EvoEngineer [Guo *et al.*, 2025], which decouples traversal techniques from population management. GPU Kernel Scientist [Andrews and Witteveen, 2025] employs a multi-stage evolutionary workflow to address the challenge of optimizing HIP kernels for the AMD accelerators. And cuPilot [Chen *et al.*, 2025a] guides evolution through high-level semantic strategies.

4.2 External Memory Management

Complex kernel optimization often requires domain-specific knowledge, such as CUDA APIs and hardware instruction sets that may be hallucinated or forgotten by the LLM. Agents in this category augment generation with relevant skills, domain knowledge, or prior interaction experiences, which serve as an external memory for LLMs. KernelEvolve [Liao *et al.*, 2025] advances the external knowledge management paradigm by integrating a sophisticated hardware-specific knowledge base specifically tailored for heterogeneous AI accelerators. Beyond retrieving unstructured textual context, recent work has explored utilizing structured representations as external memory to guide model inference. Work such as ReGraphT [Gong *et al.*, 2025] proposes a novel framework that treats a reasoning graph as a domain-specific external memory for CUDA code optimization. In this approach, the logical transitions between optimization states of large language models are externalized into a static, navigable graph structure for the small language model to retrieve. KernelBlaster [Dong *et al.*, 2026] enables agents to learn from experience by accumulating knowledge into a retrievable persistent CUDA knowledge base, EvoKernel [Zheng *et al.*, 2026] further introduces a value-driven memory and retrieved prior experiences or trajectories with different priorities. KernelSkill [Sun *et al.*, 2026] provides a dual-level memory architecture, where reusable expert skills are curated in the long-term memory.

4.3 Hardware Profiling Integration

The third dimension addresses the hardware-agnostic nature of standard LLMs by configuring the agent's persona profile with hardware specifications, and iteratively reasoning over performance profiling feedback.

QiMeng-TensorOp [Zhang *et al.*, 2025a] triggers LLMs to analyze and distill low-level hardware documentation according to user input into the generation prompt, while QiMeng-GEMM [Zhou *et al.*, 2025b] generates General Matrix Multiplication (GEMM) with the meta-prompt, which offers universal templates for various general optimization techniques and platform-specific optimization details. QiMeng-Attention [Zhou *et al.*, 2025a] considers target GPU architecture and instruction set to convert the high-level thinking language into low-level CUDA code, and implements the high-performance FlashAttention on different GPUs. SwizzlePerf [Tschand *et al.*, 2025] explicitly tackles the swizzling problem, which explicitly injects precise architectural specifications into the prompt context and restricts the search space specifically to swizzling patterns that focus solely on maximizing the L2 cache hit rate.

Complementing this, agents leverage dynamic feedback. CUDA-LLM [Chen *et al.*, 2025c] incorporates detailed target GPU specifications (e.g., warp size, cache size) into the agent's prompt. Simultaneously, compilation logs and runtime performance metrics are also aggregated to guide the optimization process. TritonForge [Li *et al.*, 2025b] utilizes profiling-guided feedback loops to iteratively analyze and identify performance bottlenecks. PRAGMA [Lei *et al.*, 2025] uses a specialized profiling module to parse low-level quantitative metrics into an interpretable natural language suggestion. KernelBand [Ran *et al.*, 2025] clusters runtime behavior of potential kernels to reduce the exploration space and utilizes profiling data as context to guide the selection of optimization strategies.

4.4 Multi-Agent Orchestration

Recognizing that kernel development inherently involves heterogeneous abilities ranging from algorithmic planning to low-level coding and debugging, recent work increasingly adopts multi-agent designs that explicitly decompose these responsibilities into coordinated roles.

STARK [Dong *et al.*, 2025] structures generation into Plan-Code-Debug phases to emulate the human team, AKG [Du *et al.*, 2025] leverages similar modularity to achieve cross-platform synthesis, while Astra [Wei *et al.*, 2025b] specializes this multi-agent approach for production-grade SGLang kernels. CudaForge [Zhang *et al.*, 2025b] employs a Coder-Judge framework driven by hardware-level feedback, whereas KForge [Sereda *et al.*, 2025] adapts this dual-agent model to new platforms using only single-shot example supervision. Addressing scale, KernelFalcon [Team and Contributors, 2024] employs a multi-agent system to tackle the challenge of GPU kernel generation of full machine learning architectures, where the system specifically addresses hierarchical task decomposition and delegation through coordinated manager and worker agents. Moreover, GEAK [Wang *et al.*, 2025] targets AMD GPUs, integrating the generator, reflector, evaluator and optimizer within a Triton-based workflow.

5 Datasets

The effectiveness of LLM-driven kernel generation depends critically on the availability of domain-specific data. Unlike general software engineering tasks, kernel generation requires models to capture hardware intrinsics, parallel execution semantics, and memory hierarchy constraints. We categorize existing resources into two groups: (1) *Training Corpora*, comprising structured datasets and kernel code repositories used for model training; and (2) *Knowledge Bases*, which typically provide domain-specific knowledge for RAG.

Data	Resource	Description	Access
I. Structured Datasets (Hugging Face & Benchmarks)			
02/2024	**The Stack v2** [Lozhkov *et al.*, 2024]	Unsupervised CUDA/Triton Corpus	[Data]
06/2024	**HPC-Instruct** [HPC-AI Tech, 2024]	Instructions for CUDA/MPI/OpenMP	[Data]
05/2025	**KernelBook** [Paliskara and Saroufim, 2025]	Torch-Triton Aligned Corpus	[Data]
02/2025	**KernelBench samples**	Kernel Code Snapshots and Profiling Data	[Data]
II. Code-Centric Corpora (GitHub Repositories)			
Layer 1: High-Performance Operator Libraries			
12/2017	**CUTLASS**	CUDA C++ Template Library for Matrix Ops	[Code]
05/2022	**FlashAttention**	Fast and Memory-Efficient Exact Attention	[Code]
11/2023	**FlagAttention**	Memory Efficient Attention Operators in Triton	[Code]
02/2024	**AoTriton**	AOT-compiled Triton kernels for AMD ROCm	[Code]
11/2021	**xFormers**	Hackable and Optimized Transformer Blocks	[Code]
08/2024	**Liger-Kernel**	Efficient Triton Kernels for LLM Training	[Code]
04/2024	**FlagGems**	Triton-based Operator Library for LLMs	[Code]
09/2022	**Bitsandbytes**	K-bit Quantization Kernels for LLMs	[Code]
09/2024	**Gemlite**	Low-Bit Matrix Multiplication Triton Kernels	[Code]
01/2025	**FlashInfer**	Kernel Library for Efficient LLM Serving	[Code]
05/2021	**FBGEMM**	Low-Precision Matrix Multiplication	[Code]
09/2022	**Transformer Engine**	Acceleration Library for Transformer Models	[Code]
09/2025	**DeepGEMM**	Clean and Efficient FP8 GEMM Kernels	[Code]
04/2026	**Tile Kernels**	A Kernel Library Written in TileLang	[Code]
Layer 2: Framework & System Integration			
10/2016	**PyTorch (ATen)**	Foundational Tensor Library for C++ and Python	[Code]
06/2023	**vLLM**	High-Efficient Serving Engine	[Code]
12/2023	**SGLang**	Structured Generation Language for LLMs	[Code]
03/2023	**llama.cpp**	LLM Inference in C/C++	[Code]
08/2023	**TensorRT-LLM**	TensorRT Toolbox for LLM Inference	[Code]
10/2019	**DeepSpeed**	System for Large Scale Model Training	[Code]
Layer 3: Domain-Specific Languages			
07/2019	**Triton**	Open-Source GPU Programming Language	[Code]
04/2024	**TileLang**	Tile-based Optimization Language	[Code]
12/2025	**cuTile**	NVIDIA's DSL for Tile-centric Programming	[Link]
III. Knowledge Bases & Educational Resources			
Documentation & Guides			
06/2007	**CUDA Guide**	CUDA C++ Programming Guide	[Docs]
06/2007	**PTX ISA**	PTX ISA Reference	[Docs]
05/2020	**Tuning Guides**	NVIDIA Architecture Tuning Guides	[Docs]
Community Indices & Tutorials			
01/2024	**GPU-MODE**	Resource Stream & KernelBook	[List]
01/2024	**Triton Index**	Community Index for Triton Optimization	[List]
06/2016	**Awesome-CUDA**	Community Curated List for CUDA	[List]
12/2023	**Awesome-GPU**	Awesome GPU Engineering List	[List]
05/2023	**LeetCUDA**	CUDA Programming Exercises	[Code]
01/2023	**Triton-Puzzles**	Puzzles for Learning Triton	[Code]
01/2011	**Colfax Research**	Technical Hub Dedicated to HPC and AI	[Link]
09/2018	**Nsight Compute**	Kernel Profiling Guide	[Docs]
07/2024	**CUDA Course**	GitHub Repo for CUDA Course	[Docs]

Table 1: A structured overview of training corpora and kernel knowledge bases. Note that the dates in the table correspond to the initial release; the libraries themselves continue to undergo active development.

Training corpora include both structured datasets and raw kernel repositories. Structured datasets typically pair high-level intents with optimized implementations, whereas repositories provide collections of expert-written kernels drawn from open-source libraries, training and inference frameworks, and domain-specific languages. Complementing executable code, knowledge bases supply hardware and optimization knowledge through technical documentation, pro-

gramming guides, tutorials, and community resources. Such knowledge can be incorporated into model pretraining or accessed dynamically through RAG. A comprehensive summary of these resources is provided in Table 1. The dates reported correspond to the initial release of each resource.

6 Benchmark

This section focuses on systematic benchmarking of kernel generation and provides a structured overview of representative evaluation benchmarks, including both evaluation metrics and benchmark datasets, to lay a solid foundation for subsequent method comparison and performance analysis.

6.1 Metrics

Kernel evaluation typically considers three aspects: correctness, performance, and composite quality metrics. Most benchmarks adopt execution-based testing, where generated kernels are executed and compared against reference CUDA or PyTorch implementations. Since kernel generation is inherently stochastic, evaluations are often repeated across multiple samples and random seeds.

Correctness measures whether a generated kernel produces outputs consistent with a reference implementation within predefined numerical tolerances. Evaluation criteria vary across kernel types and precision formats (e.g., FP16, BF16, and FP8), reflecting their distinct numerical characteristics. Detailed evaluation protocols are provided in FlashinferBench [Xing *et al.*, 2026].

Performance is primarily measured by runtime speedup relative to a reference implementation, typically using wall-clock execution time averaged over repeated trials. Some benchmarks further compare achieved performance against Speed-of-Light (SOL) estimates, which approximate the theoretical efficiency limit of the target hardware.

In addition, Efficiency refers to how effectively the generated operators utilize computation resources during execution, and Compatibility is considered when evaluating operator generation techniques across different hardware platforms or languages. Composite metrics jointly assess multiple dimensions of kernel quality. Representative examples include *Similarity*, which evaluates code similarity using lexical, syntactic, and dataflow features; and $fast_p$, which combines correctness and performance by measuring the proportion of generated kernels that are both correct and achieve a speedup greater than p:

$$\text{fast}_p = \frac{1}{N}\sum_{i=1}^{N}\mathbf{1}(correct_i \wedge \{speedup_i > p\}). \quad (1)$$

6.2 Benchmark Datasets

As summarized in Table 2, kernel benchmarks are progressively evolving toward more realistic and comprehensive evaluation. This evolution is reflected in three dimensions: *metrics*, where evaluation has expanded from correctness and runtime speedup (ParEval [Nichols *et al.*, 2024]) to comprehensive measures of efficiency, robustness, and overall kernel quality. Examples include fast_p in KernelBench, efficiency metrics in TritonBench [Li *et al.*, 2025c]; *hardware coverage*, where benchmarks increasingly extend beyond NVIDIA GPUs to encompass AMD GPUs, Huawei NPUs, and Google TPUs (such as MultiKernelBench [Wen *et al.*, 2025]); and *workload diversity*, where evaluation has shifted from simple operators toward production-grade kernels derived from real-world frameworks, systems, and AI workloads. For example, FlashInfer-Bench and SOL-ExecBench [Lin *et al.*, 2026] focus on critical kernels derived from production systems and emerging AI models, and BackendBench [Saroufim *et al.*, 2025] targets complex edge cases.

7 Challenges and Opportunities

Despite rapid progress, LLM-driven kernel generation remains a nascent research area. Advancing from proof-of-concept systems to robust and scalable deployment requires addressing challenges spanning evaluation, data, agent capabilities, infrastructure, and human-AI collaboration. This section reviews these challenges and outlines promising directions for future research.

Evaluation Reliability and Generalization. Reliable evaluation remains a fundamental challenge for LLM-driven kernel generation. Existing systems are vulnerable to reward hacking, where kernels achieve favorable benchmark scores without delivering corresponding benefits in practical deployments. At the same time, current benchmarks often cover limited workloads, hardware platforms, and execution settings, raising concerns about the generalization of reported methods. Furthermore, the impact of kernel-level improvements on end-to-end AI systems remains insufficiently understood. Future evaluation frameworks should emphasize robustness against reward hacking, broad generalization across workloads or platforms, and system-level assessment, providing a more reliable measure of real-world effectiveness.

Data Scarcity and Synthetic Scaling. Data scarcity remains a major bottleneck for LLM-driven kernel generation. High-performance kernels are sparsely represented in existing corpora, and predominantly contain final implementations while largely omitting optimization trajectories and hardware-aware expertise. As a result, critical signals for learning kernel optimization remain limited. Promising directions include large-scale kernel dataset construction, synthetic data generation, and the collection of execution-driven optimization traces. Such resources could support both model training and agentic learning, and may be essential for scaling kernel generation capabilities.

Agentic Training and Harness Engineering. Kernel optimization poses a challenging long-horizon task that requires reasoning over iterative cycles of generation, execution, profiling, and refinement. Current foundation models are not explicitly trained for such long-horizon optimization trajectories, while existing agentic systems often rely on handcrafted workflows that struggle with exploration efficiency, context management, and long-term credit assignment. Future progress may require advances in both long-horizon agentic training and autonomous harness engineering. Moreover, recent advances in general-purpose coding agents (e.g.,

Name	Time	Metrics	Hardware	Description
ParEval	01/2024	C S E	N A	420 expert-selected tasks across 12 algorithmic domains for benchmarking general parallel code generation.
KernelBench	02/2025	C f	N	250 PyTorch-to-CUDA kernel generation tasks, curated from popular GitHub repositories and official PyTorch operators, for evaluating AI/DL kernel generation.
TritonBench	02/2025	C S S E*	N	TritonBench evaluates Triton kernel generation via two subsets: 184 high-level kernels sourced from popular GitHub projects (TritonBench-G) and 166 fusion tasks derived from diverse PyTorch operators with different frequencies of usage (TritonBench-T).
MultiKernel-Bench	07/2025	C S	H N G	285-task benchmark across 14 operator categories for multi-platform DL kernel synthesis.
TritonBench-revised & ROCm Benchmark	07/2025	C S	A	An AMD GPU-centric evaluation dataset comprising 30 expert-verified ROCm kernels and an adapted version of TritonBench-G, specifically optimized for AMD GPU performance benchmarking.
Robust-kbench	09/2025	C S	N	A robustness-focused benchmark featuring 9 deep learning task categories, derived by refining and extending KernelBench.
BackendBench	09/2025	C S	N	A rigorous evaluation framework that enforces PyTorch's official core library standards. Current use cases primarily leverage NVIDIA CUDA and Triton, yet the architecture remains backend-agnostic.
CUDAEval	10/2025	C S	N	Leveraging 313 curated tasks from the Stack v2 to benchmark the efficacy of reasoning transfer in CUDA code optimization.
FlashInfer-Bench	01/2026	C f	N	Provide a unified schema describing kernel definitions, workloads, implementations, and evaluations, including eight representative kernel types used in LLM inference.
SOL-ExecBench	03/2026	SOL Score	N	A benchmark for GPU kernel optimization built around hardware Speed-of-Light (SOL) targets.

* Efficiency here is defined as the ratio of the operator's measured throughput to the theoretical maximum performance.

Table 2: Benchmark datasets for kernel generation and optimization. Metrics: C Correctness, S Speedup, E Efficiency, f $fast_p$, S Similarity. Hardware Platforms: N NVIDIA GPUs, H HUAWEI NPUs, G Google TPUs, A AMD GPUs.

Claude Code, OpenCode) suggest that kernel optimization may not require developing task-specific agents from scratch. Instead, foundation agents can be specialized through harness engineering, leveraging domain-specific environments, tools, and feedback to enable scalable agentic performance engineering.

Scalable Infrastructure for Synthesis and Training. Scalable infrastructure remains a critical bottleneck for large-scale data synthesis and agentic training. Key challenges include building distributed and securely isolated sandbox environments for execution, addressing latency mismatches between agent rollout and kernel compilation and verification, and designing fault-tolerant, multi-device services that ensure stable and efficient training at scale. Advances in such infrastructure are essential for transforming kernel synthesis and data sampling from low-throughput, ad-hoc experimentation into a systematic, data-driven evolutionary process, enabling continuous improvement of agentic kernel optimization systems.

Human-AI Collaboration for Kernel Generation. Human-AI collaboration represents a promising alternative to fully autonomous kernel optimization. A central challenge is to establish effective feedback loops between human expertise and agentic exploration. While human-in-the-loop guidance can guide optimization through high-level objectives and constraints, human-from-the-loop learning enables knowledge discovered by agents to be transferred back to developers. Given the verifiable nature of kernel optimization, such bidirectional collaboration may create an effective cycle of human and agent co-evolving, expanding the frontier of automated performance engineering.

8 Conclusion

This survey highlights the transformative potential of LLMs and agentic systems for automating kernel generation and optimization, synthesizing recent advances in methodologies alongside the development of kernel-centric datasets and benchmarks. Looking ahead, future progress will depend on more reliable evaluation protocols, as well as the data infrastructure and harness engineering required for agentic kernel generation. These advances have the potential not only to alleviate the burden of manual kernel engineering, but also to unlock substantial productivity gains for rapidly scaling AI infrastructure.

References

[Andrews and Witteveen, 2025] Martin Andrews and Sam Witteveen. Gpu kernel scientist: An llm-driven framework for iterative kernel optimization. In *ICML Workshop*, 2025.

[BAAI, 2025] BAAI. KernelGen. https://github.com/flagos-ai/KernelGen, 2025.

[Baronio *et al.*, 2025] Carlo Baronio, Pietro Marsella, et al. Kevin: Multi-turn rl for generating cuda kernels. *arXiv preprint arXiv:2507.11948*, 2025.

[Cao *et al.*, 2026a] Shiyi Cao, Ziming Mao, et al. K-search: Llm kernel generation via co-evolving intrinsic world model. *arXiv preprint arXiv:2602.19128*, 2026.

[Cao *et al.*, 2026b] Xinzi Cao, Jianyang Zhai, et al. Ascendkernelgen: A systematic study of llm-based kernel generation for neural processing units. *arXiv preprint arXiv:2601.07160*, 2026.

[Chen *et al.*, 2025a] Jinwu Chen, Qidie Wu, et al. cupilot: A strategy-coordinated multi-agent framework for cuda kernel evolution. *arXiv preprint arXiv:2512.16465*, 2025.

[Chen *et al.*, 2025b] Terry Chen, Bing Xu, et al. Automating gpu kernel generation with deepseek-r1 and inference time scaling. NVIDIA Developer Blog, 2025.

[Chen *et al.*, 2025c] Wentao Chen, Jiace Zhu, et al. Cuda-llm: Llms can write efficient cuda kernels. *arXiv preprint arXiv:2506.09092*, 2025.

[Choquette *et al.*, 2021] Jack Choquette, Wishwesh Gandhi, et al. Nvidia A100 tensor core gpu: Performance and innovation. *IEEE Micro*, 41(2):29–35, 2021.

[Dai *et al.*, 2026] Weinan Dai, Hanlin Wu, et al. Cuda agent: Large-scale agentic rl for high-performance cuda kernel generation. *arXiv preprint arXiv:2602.24286*, 2026.

[Dong *et al.*, 2025] Juncheng Dong, Yang Yang, et al. Stark: Strategic team of agents for refining kernels. *arXiv preprint arXiv:2510.16996*, 2025.

[Dong *et al.*, 2026] Kris Shengjun Dong, Sahil Modi, et al. Kernelblaster: Continual cross-task cuda optimization via memory-augmented in-context reinforcement learning. *arXiv preprint arXiv:2602.14293*, 2026.

[Du *et al.*, 2025] Jinye Du, Quan Yuan, et al. Akg kernel agent: A multi-agent framework for cross-platform kernel synthesis. *arXiv preprint arXiv:2512.23424*, 2025.

[Du *et al.*, 2026] He Du, Qiming Ge, et al. Kernel-smith: A unified recipe for evolutionary kernel optimization. *arXiv preprint arXiv:2603.28342*, 2026.

[Fisches *et al.*, 2025] Zacharias V. Fisches, Sahan Paliskara, et al. Kernelllm: Making kernel development more accessible, 6 2025.

[Gong *et al.*, 2025] Junfeng Gong, Zhiyi Wei, et al. From large to small: Transferring cuda optimization expertise via reasoning graph. *arXiv preprint arXiv:2510.19873*, 2025.

[Guo *et al.*, 2025] Ping Guo, Chenyu Zhu, et al. Evoengineer: Mastering automated cuda kernel code evolution with large language models. *arXiv preprint arXiv:2510.03760*, 2025.

[Hammond *et al.*, 2025] Alec M Hammond, Aram Markosyan, et al. Agentic operator generation for ML ASICs. *arXiv preprint arXiv:2512.10977*, 2025.

[HPC-AI Tech, 2024] HPC-AI Tech. hpc-instruct: A dataset for hpc instruction tuning. https://huggingface.co/datasets/hpcgroup/hpc-instruct, 2024. Hugging Face Dataset.

[Jaber and Jaber, 2026] Jaber Jaber and Osama Jaber. Autokernel: Autonomous gpu kernel optimization via iterative agent-driven search. *arXiv preprint arXiv:2603.21331*, 2026.

[Kaplan *et al.*, 2020] Jared Kaplan, Sam McCandlish, et al. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[Kong *et al.*, 2025] Lingcheng Kong, Jiateng Wei, et al. Concur: Conciseness makes state-of-the-art kernel generation. *CoRR*, abs/2510.07356, 2025.

[Lange *et al.*, 2025] Robert Tjarko Lange, Qi Sun, et al. Towards robust agentic cuda kernel benchmarking, verification, and optimization. *arXiv preprint arXiv:2509.14279*, 2025.

[Lei *et al.*, 2025] Kelun Lei, Hailong Yang, et al. Pragma: A profiling-reasoned multi-agent framework for automatic kernel optimization. *arXiv preprint arXiv:2511.06345*, 2025.

[Li *et al.*, 2025a] Annan Li, Chufan Wu, et al. The fm agent. *arXiv preprint arXiv:2510.26144*, 2025.

[Li *et al.*, 2025b] Haonan Li, Keyu Man, et al. Tritonforge: Profiling-guided framework for automated triton kernel optimization. *arXiv preprint arXiv:2512.09196*, 2025.

[Li *et al.*, 2025c] Jianling Li, ShangZhan Li, et al. Tritonbench: Benchmarking large language model capabilities for generating triton operators. In *ACL*, pages 23053–23066, 2025.

[Li *et al.*, 2025d] Shangzhan Li, Zefan Wang, et al. Autotriton: Automatic triton programming with reinforcement learning in llms. *arXiv preprint arXiv:2507.05687*, 2025.

[Liao *et al.*, 2021] Heng Liao, Jiajin Tu, et al. Ascend: a scalable and unified architecture for ubiquitous deep neural network computing: Industry track paper. In *HPCA*, pages 789–801. IEEE, 2021.

[Liao *et al.*, 2025] Gang Liao, Hongsen Qin, et al. Kernelevolve: Scaling agentic kernel coding for heterogeneous ai accelerators at meta. *arXiv preprint arXiv:2512.23236*, 2025.

[Lin *et al.*, 2026] Edward Lin, Sahil Modi, et al. Solexecbench: Speed-of-light benchmarking for real-world gpu kernels against hardware limits. *arXiv preprint arXiv:2603.19173*, 2026.

[Liu *et al.*, 2026] Wei Liu, Jiawei Xu, et al. Dr. kernel: Reinforcement learning done right for triton kernel generations. *arXiv preprint arXiv:2602.05885*, 2026.

[Lozhkov *et al.*, 2024] Anton Lozhkov, Raymond Li, et al. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*, 2024.

[Nichols *et al.*, 2024] Daniel Nichols, Joshua H. Davis, et al. Can large language models write parallel code? In *HPDC*, page 281–294, 2024.

[Ou *et al.*, 2026] Jiefu Ou, Sapana Chaudhary, et al. Maxcode: A max-reward reinforcement learning framework for automated code optimization. *arXiv preprint arXiv:2601.05475*, 2026.

[Ouyang *et al.*, 2025] Anne Ouyang, Simon Guo, et al. Kernelbench: Can LLMs write efficient GPU kernels? In *ICML*, 2025.

[Paliskara and Saroufim, 2025] Sahan Paliskara and Mark Saroufim. Kernelbook, 5 2025.

[Ran *et al.*, 2025] Dezhi Ran, Shuxiao Xie, et al. Kernelband: Boosting llm-based kernel optimization with a hierarchical and hardware-aware multi-armed bandit. *arXiv preprint arXiv:2511.18868*, 2025.

[Saroufim *et al.*, 2025] Mark Saroufim, Jiannan Wang, et al. Backendbench: An evaluation suite for testing how well llms and humans can write pytorch backends, 2025.

[Sereda *et al.*, 2025] Taras Sereda, Tom St John, et al. Kforge: Program synthesis for diverse ai hardware accelerators. *arXiv preprint arXiv:2511.13274*, 2025.

[Su *et al.*, 2025] Songqiao Su, Xiaofei Sun, et al. Cuda-l2: Surpassing cublas performance for matrix multiplication through reinforcement learning. *arXiv preprint arXiv:2512.02551*, 2025.

[Sun *et al.*, 2026] Qitong Sun, Jun Han, et al. Kernelskill: A multi-agent framework for gpu kernel optimization. *arXiv preprint arXiv:2603.10085*, 2026.

[Tariq *et al.*, 2025] Muhammad Usman Tariq, Abhinav Jangda, et al. Peak: A performance engineering ai-assistant for gpu kernels powered by natural language transformations. *arXiv preprint arXiv:2512.19018*, 2025.

[Team and Contributors, 2024] PyTorch Team and Contributors. Kernelfalcon: Autonomous GPU kernel generation via deep agents, 2024. Accessed: 2026-01-02.

[Tschand *et al.*, 2025] Arya Tschand, Muhammad Awad, et al. Swizzleperf: Hardware-aware llms for gpu kernel performance optimization. *arXiv preprint arXiv:2508.20258*, 2025.

[Vaswani *et al.*, 2017] Ashish Vaswani, Noam Shazeer, et al. Attention is all you need. *NeurIPS*, 30, 2017.

[Wang *et al.*, 2024] Lei Wang, Chen Ma, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, 2024.

[Wang *et al.*, 2025] Jianghui Wang, Vinay Joshi, et al. Geak: Introducing triton kernel ai agent & evaluation benchmarks. *arXiv preprint arXiv:2507.23194*, 2025.

[Wang *et al.*, 2026] Yaoyu Wang, Hankun Dai, et al. Mastering sparse CUDA generation through pretrained models and deep reinforcement learning. In *ICLR*, 2026.

[Wei *et al.*, 2025a] Anjiang Wei, Allen Nie, et al. Improving parallel program performance with llm optimizers via agent-system interfaces. In *ICML*, 2025.

[Wei *et al.*, 2025b] Anjiang Wei, Tianran Sun, et al. Astra: A multi-agent system for gpu kernel performance optimization. *arXiv preprint arXiv:2509.07506*, 2025.

[Wen *et al.*, 2025] Zhongzhen Wen, Yinghui Zhang, et al. Multikernelbench: A multi-platform benchmark for kernel generation. *arXiv eprints, pp. arXiv–2507*, 2025.

[Woo *et al.*, 2025] Jiin Woo, Shaowei Zhu, et al. Tritonrl: Training llms to think and code triton without cheating. *arXiv preprint arXiv:2510.17891*, 2025.

[Wu, 2023] Peng Wu. Pytorch 2.0: The journey to bringing compiler technologies to the core of pytorch (keynote). In *CGO*, 2023.

[Xie *et al.*, 2026] Shuxiao Xie, Shuyang Xie, et al. AKO: Agentic kernel optimization. https://tongminglaic.github.io/AKO, 2026. Technical report.

[Xing *et al.*, 2026] Shanli Xing, Yiyan Zhai, et al. Flashinfer-bench: Building the virtuous cycle for ai-driven llm systems. *arXiv preprint arXiv:2601.00227*, 2026.

[Yang *et al.*, 2026] Jian Yang, Wei Zhang, et al. Incoder-32b: Code foundation model for industrial scenarios. *arXiv preprint arXiv:2603.16790*, 2026.

[Zhang *et al.*, 2025a] Xuzhi Zhang, Shaohui Peng, et al. Qimeng-tensorop: Automatically generating high-performance tensor operators with hardware primitives. *arXiv preprint arXiv:2505.06302*, 2025.

[Zhang *et al.*, 2025b] Zijian Zhang, Rong Wang, et al. Cudaforge: An agent framework with hardware feedback for cuda kernel optimization. *arXiv preprint arXiv:2511.01884*, 2025.

[Zheng *et al.*, 2026] Yujie Zheng, Zhuo Li, et al. Towards cold-start drafting and continual refining: A value-driven memory approach with application to npu kernel synthesis. *arXiv preprint arXiv:2603.10846*, 2026.

[Zhou *et al.*, 2025a] Qirui Zhou, Shaohui Peng, et al. QiMeng-attention: SOTA attention operator is generated by SOTA attention algorithm. In *ACL*, 2025.

[Zhou *et al.*, 2025b] Qirui Zhou, Yuanbo Wen, et al. Qimeng-gemm: Automatically generating high-performance matrix multiplication code by exploiting large language models. In *AAAI*, 2025.

[Zhu *et al.*, 2025] Xinguo Zhu, Shaohui Peng, et al. Qimeng-kernel: Macro-thinking micro-coding paradigm for llm-based high-performance gpu kernel generation. *arXiv preprint arXiv:2511.20100*, 2025.

[Zhu *et al.*, 2026] Haowei Zhu, Puyuan Yang, et al. Diffbench meets diffagent: End-to-end llm-driven diffusion acceleration code generation. *arXiv preprint arXiv:2601.03178*, 2026.