TOWARDS AUTOMATED KERNEL GENERATION IN THE ERA OF LLMS: A SURVEY

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,*], **Yonghua Lin**[1]

[1]BAAI, [2]BNU, [3]PKU, [4]BIT, [5]CU, [6]BJTU, [7]RUC, [8]HKUST(GZ)

ABSTRACT

The performance of modern AI systems is fundamentally constrained by the quality of their underlying GPU kernels, which dominate the cost of large-scale training and inference. Achieving near-optimal kernels requires expert-level understanding of GPU architectures, programming models, and intricate hardware–software trade-offs, making kernel engineering a critical but notoriously time-consuming and non-scalable process. Recent advances in large language models (LLMs) and agentic LLM systems have opened new possibilities for automating kernel generation and optimization. LLMs are well-suited to compress expert-level kernel engineering knowledge that is difficult to formalize and reuse, while agentic LLM systems further enable scalable optimization by casting kernel development as an iterative, feedback-driven loop involving code generation and performance-guided refinement. Together, they offer a practical pathway toward scalable and automated kernel engineering. Rapid progress has been made in this area. However, the field remains fragmented, lacking 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 compiling 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.

Keywords Large language models · LLM-based agents · Kernel generation

1 Introduction

The rapid scaling of large language models (LLMs) has led to unprecedented computational demands, making efficient hardware utilization a central challenge in modern AI systems [1, 2, 3, 4, 5, 6]. To meet these demands, specialized accelerators such as GPUs, NPUs, and TPUs have become the backbone of large-scale training and inference [7, 8, 9, 10]. At the core of these platforms are kernels that implement fundamental operations, including matrix multiplication and attention, which dominate execution time in LLM workloads. As a result, the end-to-end performance, efficiency, and cost of LLM systems are largely determined by kernel efficiency rather than hardware peak capability [11, 12]. Even small improvements in kernel performance can therefore yield significant gains in runtime, energy efficiency, and scalability, underscoring kernels as a critical factor in the continued advancement of large-scale AI [13, 14].

Despite their foundational importance, developing efficient kernels remains a formidable engineering challenge. Achieving near-peak utilization demands deep expertise in both algorithmic structure and hardware-specific details, including parallel execution models [15], memory hierarchies [16], and fine-grained resource management [17]. Kernel optimization is inherently non-scalable: implementations are often tightly coupled to specific hardware architectures and workload characteristics, making them difficult to reuse or generalize [18, 19]. Porting an algorithm across GPU generations or hardware vendors typically requires extensive manual retuning, making kernel development a labor-

*Corresponding author, liuguang@baai.ac.cn

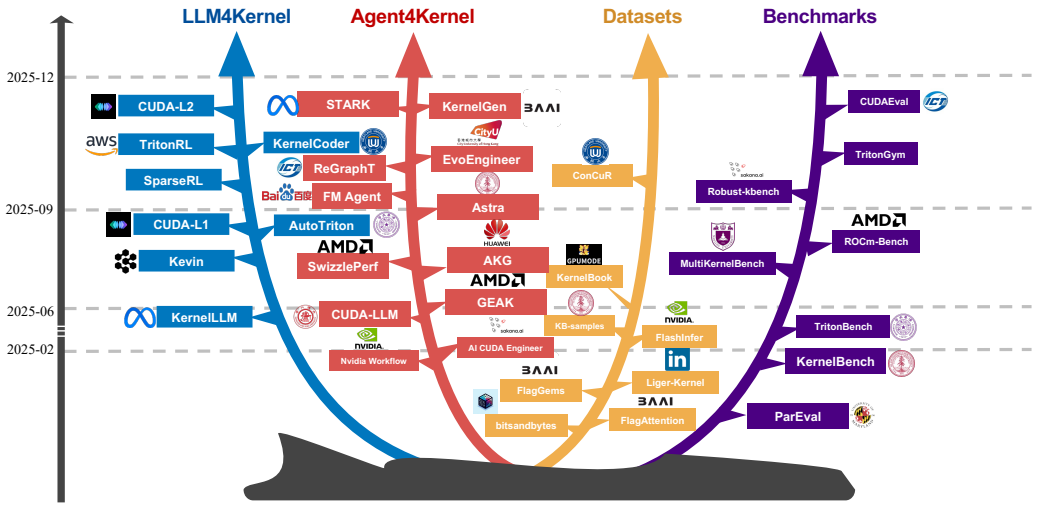

Figure 1: The Framework of the survey.

intensive and error-prone endeavor, even in the context of decades of improvements in GPU software ecosystems [20, 21].

In response to these challenges, LLMs and LLM-based agents offer a transformative paradigm for kernel generation. By training on vast repositories of code and documentation, LLMs effectively compress expert-level "world knowledge" regarding hardware specifications and programming patterns, enabling them to bridge the semantic gap between high-level algorithms and low-level implementation details [22, 23]. Beyond static code generation, LLM-based agents excel in navigating the irregular optimization landscape through iterative refinement. By leveraging real-world execution feedback ranging from compiler diagnostics to runtime profiling, these agents can autonomously hypothesize, verify, and refine kernels [24, 25]. This closed-loop approach not only drastically reduces the engineering but also generalizes across workloads and hardware configurations points, toward a future of scalable, automated kernel discovery. As a result, LLMs and LLM-based agents are emerging as compelling foundations for the next generation of kernel generation and optimization frameworks.

The integration of LLMs and LLM-based agents into kernel generation marks a rapidly advancing frontier in AI systems research. Progress on two fronts has been particularly notable: core LLM capabilities have been significantly boosted through data synthesis, supervised fine-tuning, and reinforcement learning, while agent frameworks have matured in key areas such as strategic planning, output verification, and memory-augmented reasoning. However, the absence of a systematic survey has resulted in a fragmented research landscape, lacking clear definitions of foundational paradigms, standardized benchmarks, and a coherent road-map for future work. This survey addresses this gap by presenting a unified overview of the field, clarifying foundational concepts, and highlighting emergent methodologies trends. A key contribution is our consolidated resource infrastructure, featuring a structured compilation 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 prior art, we also identify 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

2.1 Large Language Models

The foundation of modern LLMs is the Transformer architecture [26], which revolutionized sequence modeling by replacing recurrence with self-attention mechanisms. Formally, for a query Q, key K, and value V, the attention output

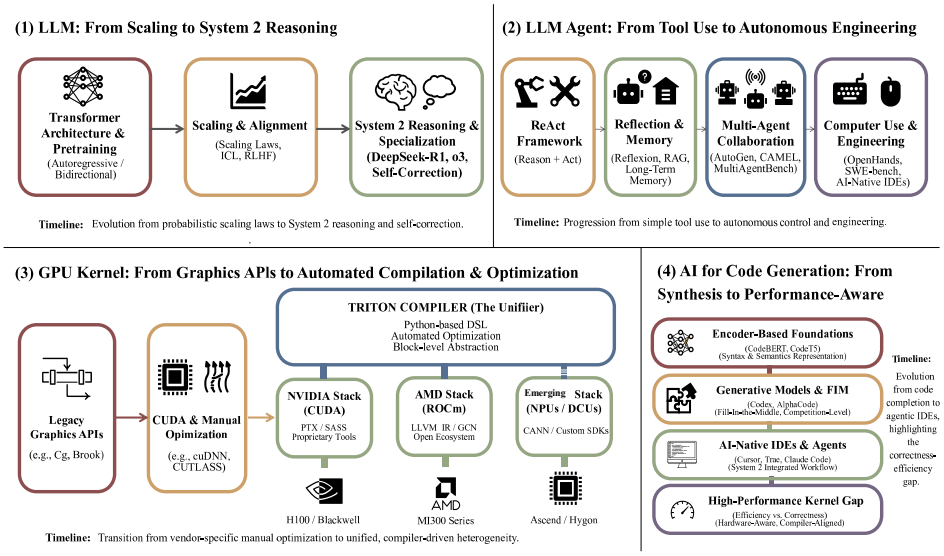

Figure 2: The Illustration of LLM, Agents and Kernels.

is computed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where d_k is the dimension of the keys. Fundamentally, these architectures function as probabilistic predictors trained via the Next Token Prediction (NTP) objective [27]. 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.

The evolution of LLMs has traversed three pivotal phases. The first phase focused on scale and emergence: as codified by scaling laws [1], increasing parameters unlocked In-Context Learning (ICL) [28], allowing models like GPT-3 to adapt to new tasks via prompts without weight updates. The second phase prioritized alignment: techniques such as Reinforcement Learning from Human Feedback (RLHF) [3, 29] were introduced to steer models toward human intent, culminating in the robust capabilities of GPT-4 [30] and the open Llama ecosystem [31].

By 2025, the landscape has expanded into a third phase of specialization and depth. First, reasoning-intensive models have emerged to tackle complex logic. DeepSeek-R1 [5] and OpenAI o3 [32] utilize large-scale reinforcement learning to internalize "System 2" thinking[33], enabling the rigorous self-correction essential for kernel implementation. Second, general-purpose foundation models continue to scale. GPT-5 [34] and DeepSeek-V3 [35] have pushed the boundaries of general knowledge, while Claude 4 [36] has established itself as a premier expert in software engineering. Finally, the ecosystem is bolstered by long-context models like Gemini 2.0 [37] for processing extensive hardware documentation, and powerful open-weight models such as Llama 4 and Qwen-Coder [38, 39], which democratize access for localized code generation tools.

2.2 LLM-based Autonomous Agents

While LLMs serve as the cognitive engine, autonomous agents extend this capability by incorporating planning, memory, and tool usage to interact with environments—a transition from reasoning to autonomy comprehensively reviewed in [40]. A widely adopted formulation defines an agent as: Agent = LLM + Planning + Memory + Tools [41]. In this framework, the LLM functions as the "brain", orchestrating actions through reasoning strategies like ReAct [42], which interleaves chain-of-thought reasoning with execution. To overcome the static nature of pre-trained weights, agents utilize Tool Use mechanisms [43], enabling them to invoke external APIs—such as compilers or interpreters—to perform actions beyond the model's internal knowledge.

As agent architectures matured, research focused on enhancing robust execution and collaboration. To support long-horizon tasks, frameworks like Reflexion [44] introduced verbal reinforcement learning, allowing agents to self-correct by analyzing past failures. Concurrently, memory architectures evolved from simple context windows to sophisticated retrieval mechanisms [45], enabling agents to maintain coherent long-term behaviors within generative simulations [46]. Furthermore, single-agent limitations were addressed by Multi-Agent Systems (MAS). Frameworks such as AutoGen [47] and CAMEL [48] demonstrated that coordinating specialized agents via role-playing conversation significantly improves performance, a capability now rigorously quantified by dedicated evaluation platforms like MultiAgentBench [49].

Contemporary advancements have accelerated the deployment of agents in specialized domains. Broad surveys [50] highlight a shift from generalist assistants to expert systems in fields ranging from biomedicine, exemplified by GeneAgent [51], to autonomous engineering. In the latter domain, paradigms like Anthropic's Computer Use [52] and OpenHands [53] have matured to allow agents to perceive screens and control operating systems directly. This capability enables agents to navigate complex development environments, utilizing benchmarks like SWE-bench [54] to drive the development of systems that combine retrieval-augmented generation with repository-level navigation [55].

2.3 Kernel Programming

General-purpose computation initially relied on restrictive graphics APIs [56, 57, 58] until NVIDIA's CUDA [59, 60, 61] revolutionized the field by transforming GPUs into programmable platforms [62, 63]. This shift established a robust ecosystem of high-performance libraries, such as cuBLAS, cuDNN, and CUTLASS[64, 65, 66, 67], though achieving peak efficiency has become increasingly difficult due to the complexity of classical tiling techniques [68, 69] and modern hardware features like Tensor Cores [70, 71, 72, 73]. The engineering effort required to saturate these asynchronous units, as evidenced by the optimization challenges in FlashAttention-3 [14], has significantly raised the expertise barrier, further entrenching implementations within vendor-specific execution models.

As GPU computing expanded beyond a single-vendor ecosystem, kernel programming began to face the challenge of performance portability across heterogeneous accelerators. The slowdown of Moore's Law and the rise of domain-specific architectures have led to a proliferation of hardware platforms [74, 75], including AMD accelerators with the ROCm stack [76], Intel's oneAPI ecosystem [77], and a growing number of AI accelerators from vendors such as Huawei Ascend, Cambricon, Moore Threads and Iluvatar CoreX [10, 78, 79, 80]. Although many of these platforms expose CUDA-like programming interfaces to ease migration, substantial differences in execution models, memory hierarchies, and supported hardware features often result in widely divergent performance. Consequently, kernels optimized for one device frequently transfer poorly to others despite nominal API compatibility, highlighting a central limitation of traditional kernel programming models: language-level portability does not imply performance portability [81, 82].

Against this backdrop, Triton [21] emerged as a transformative paradigm for GPU kernel development. Introduced by OpenAI, Triton provides a Python-based, block-level domain-specific language that raises the abstraction above CUDA by allowing developers to express tile-level computation while delegating hardware-specific mapping and optimization to the compiler [83]. This design decouples kernel semantics from low-level execution details, enabling better retargeting across architectures and motivating its adoption as the primary backend of PyTorch 2.0's TorchInductor [84, 85]. Building on this foundation, the broader kernel programming ecosystem continues to evolve in multiple directions: systems such as JAX/Pallas [86, 87] and Helion [88] emphasize tighter integration with high-level frameworks, while extensions like TLX [89], Gluon [90], and TileLang [91] explore greater expressiveness or composability within compiler-driven workflows. In parallel, NVIDIA has introduced CUDA Tile and its Python interface cuTile [92], which expose a tile-centric programming model within the CUDA ecosystem to simplify tensor-core-oriented kernel development while retaining close alignment with vendor-specific execution models. Together, these approaches reflect a broader shift toward higher-level, compiler-mediated abstractions as a response to growing architectural complexity and the need for scalable performance portability across heterogeneous accelerators.

2.4 AI for Code Generation

Code generation applies the probabilistic modeling of LLMs to programming languages, a domain comprehensively surveyed in [93]. Fundamentally, this process treats code as a structured sequence. Prior to modern decoder-only models, encoder-based architectures like CodeBERT [94] and CodeT5 [95] established strong foundations by learning robust representations of code syntax and semantics. Building on this, generative models utilize techniques like Fill-In-the-Middle (FIM) [96] to handle non-sequential dependencies. Milestones such as AlphaCode [97] demonstrated human-level performance in competitive programming, while open-source initiatives like CodeGen [98] laid the groundwork for the democratization of coding assistants initiated by OpenAI's Codex [99].

Entering 2025, the field has evolved from simple code completion to fully integrated AI-native development environments. Tools like Cursor [100] and Trae Agent [101] have redefined the developer experience by embedding autonomous "System 2" reasoning directly into the IDE workflow. Furthermore, agentic command-line interfaces such as Claude code [102] allow models to autonomously navigate file systems, execute tests, and refactor complex repositories. These advancements demonstrate a shift from generating isolated snippets to managing large-scale software engineering tasks.

However, a critical distinction exists between general software engineering and high-performance kernel generation. General code generation prioritizes functional correctness, typically agnostic to the underlying hardware. In contrast, kernel optimization aligns closer to compiler heuristic discovery [103] and graph-based dataflow analysis [104], demanding execution efficiency and strict hardware adaptation. A functionally correct matrix multiplication kernel is of little value if it utilizes only a fraction of the GPU's throughput, and efficient kernels must be tailored to specific heterogeneous architectures. Consequently, generating operators requires a specialized paradigm that transcends simple correctness to optimize for latency, memory bandwidth, and micro-architectural constraints.

3 LLM for Kernels Generation

Building on advances in AI-driven code generation, recent work has increasingly applied large language models to the synthesis of high-performance kernels, spanning programming environments that range from low-level systems languages such as CUDA to higher-level domain-specific frameworks such as Triton. Although these environments differ substantially in abstraction and expressiveness, they share core challenges involving correctness, parallel decomposition, memory management, and performance-sensitive algorithmic structuring. To highlight the methodological patterns that have emerged across this landscape, the following sections review two principal families of post-training techniques used to specialize LLMs for kernel generation: *supervised fine-tuning* and *reinforcement learning*. These categories capture the dominant approaches explored in current research and illustrate how common ideas and training principles are beginning to shape kernel synthesis across diverse programming abstractions.

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. One influential line of work shows that the structure and clarity of model reasoning can strongly affect kernel correctness and performance. ConCuR [105] demonstrates this by constructing a curated dataset in which training samples are selected based on the conciseness of their reasoning processes, the speedup they achieve, and the diversity of their computational tasks. Fine-tuning on such data leads to KernelCoder, a model capable of generating CUDA kernels with state-of-the-art reliability and efficiency. Another direction builds paired training corpora through compiler alignment, where kernel implementations are automatically generated to mirror high-level operators. KernelLLM [22] adopts this strategy by using the Triton compiler to produce aligned PyTorch–Triton examples and by applying instruction tuning with structured prompts that explicitly encode the mapping between computation and kernel structure. Together, these approaches show that well-designed supervised datasets—whether curated through reasoning quality or constructed through compiler-generated alignment—can effectively specialize LLMs for robust and high-performance GPU kernel synthesis across different programming abstractions.

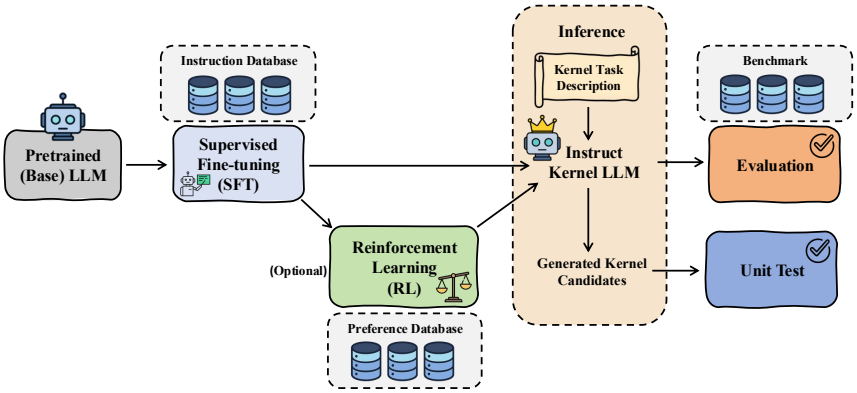

Figure 3: The category of LLM-based methods for kernel generation.

3.2 Reinforcement Learning

Reinforcement learning (RL) has emerged as a powerful paradigm for improving kernel generation by enabling iterative optimization driven by execution-based and verification-based feedback. A central advantage of RL in this context is its ability to directly incorporate signals reflecting compilation success, runtime behavior, and output correctness, thereby aligning model optimization with practical performance objectives. SparseRL [106] exemplifies this approach by using compilation outcomes and execution characteristics as reward signals, leading to substantial improvements in sparse matrix–vector and sparse matrix–matrix multiplication tasks. Kevin [23] further formulates kernel synthesis as a multi-turn optimization process, in which the model repeatedly proposes candidate kernels, receives feedback from compilation and runtime evaluation, and refines subsequent generations accordingly. To address the challenge of long-horizon credit assignment, this framework introduces cross-turn reward attribution and context compression mechanisms, enabling effective optimization over extended interaction trajectories.

Recent advances also emphasize the importance of robust reward design and verifiable evaluation, particularly in settings where valid kernel generation is initially rare and reward signals are sparse. AutoTriton [107] addresses this issue by combining structural assessments of generated kernels with execution-based runtime rewards, encouraging improvements in both algorithmic validity and empirical efficiency. TritonRL [108] extends this line of work through hierarchical reward decomposition and explicit verification of code outputs and intermediate reasoning traces, reducing susceptibility to reward hacking and promoting more faithful optimization behavior. CUDA-L1 [109] employs contrastive reinforcement learning to compare multiple candidate kernels and infer relative performance differences, while introducing an LLM-as-a-judge component that critiques intermediate outputs and provides dense, informative feedback when compilation or execution fails. Further advancing this approach, CUDA-L2 [110] builds upon CUDA-L1 by introducing more sophisticated reinforcement learning techniques tailored specifically for matrix multiplication optimization. By leveraging a combination of advanced RL algorithms and domain-specific heuristics, CUDA-L2 consistently demonstrates superior performance compared to the widely-used cuBLAS library, which is the industry standard for GPU-accelerated linear algebra operations.

4 LLM Agent for Kernels Generation

Although foundational LLMs encode substantial knowledge of kernels and hardware optimization, relying on them alone typically reduces kernel development to a static, one-pass inference process, in which code is generated without systematic verification or iterative refinement. 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 that would be prohibitively costly with purely manual or single-pass LLM-based methods. To systematically evaluate how agentic abilities are applied to kernel generation and optimization, we categorized recent agent-driven advancements into four structural dimensions: *learning mechanisms*, *external memory management*, *hardware profiling integration* and *multi-agent orchestration*.

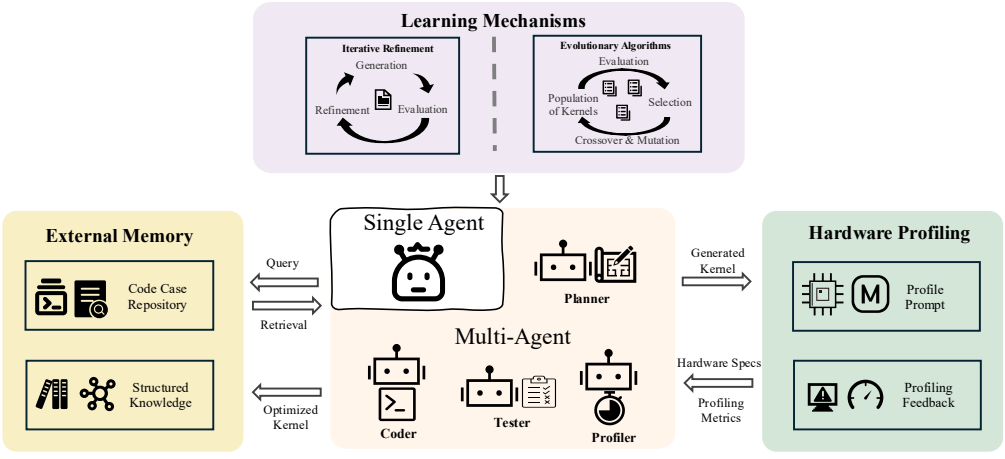

Figure 4: The category of LLM Agent-based methods for kernel generation.

4.1 Learning Mechanisms

The first dimension of advancement focuses on the learning and search strategy employed during the generation process. Early approaches treated kernel generation as a translation task, but recent works frame it as a search problem over the program space, utilizing iterative refinement and evolutionary algorithms to navigate the non-convex optimization landscape. Several systems employ a "generate-evaluate-refine" loop. Caesar (referenced as an extended baseline in KernelBench [111]) utilizes a single-agent loop that refines kernels based on simple execution feedback. Here the feedback context consists of the model's previous generation, the compiler and execution feedback, as well as the profiler output if the generation is compiled and executed successfully. Similarly, techniques exploring Inference-Time Scaling [112] demonstrate that scaling the number of samples and iteratively refining solutions significantly improves success rates by trading test-time compute for code quality. In their research, it was observed that allocating more than 10 minutes per problem in the Level-1 category of KernelBench empowers the iterative refinement workflow to generate numerically correct code for the majority of the 100 problems. Moreover, KernelGen [113] also leverages test-time scaling and reflection techniques to enable kernel generation for multi-chip backends

Furthermore, to escape local optima inherent in linear refinement, recent frameworks adopt population-based evolutionary methods. Lange et al. [114] introduce a robust agentic framework that utilizes an evolutionary meta-generation procedure. The agent translates PyTorch code to CUDA and then iteratively optimize runtime via mutation and crossover operations. Specifically, three verifier tuned for compilation, memory and numerical accuracy will help filter initial seed kernels before evolution. Similarly, FM Agent [115] proposes a two-stage framework combining a "Cold-Start" phase guided by expert initialization, followed by an "Evolutionary Stage" with the principles of diversity preservation, adaptive evolution, and multi-population dynamics. This approach maintains population diversity to discover novel optimization patterns, achieving state-of-the-art results on the KernelBench. Additionally, EvoEngineer [116] formalizes kernel optimization as a constrained discrete text-search problem and orthogonally decouples LLM-driven code evolution into two axes: traverse techniques that govern how the space is explored, and population management that governs how candidate solutions are maintained and updated. In addition to studies for CUDA hardware, GPU Kernel Scientist [117] addresses the challenge of optimizing HIP kernels for AMD accelerators AMD MI300, the newer or sparsely documented GPU architectures where traditional developer tooling is limited. The framework employs large language models in a multi-stage evolutionary workflow: it selects promising prior kernel variants, synthesizes optimization hypotheses grounded in code analysis and AMD documentation, and autonomously implements and submits these candidates for external performance evaluation.

4.2 External Memory Management

Complex kernel optimization often requires domain-specific knowledge (e.g., obscure CUDA APIs, hardware instruction sets) that may be hallucinated or forgotten by the LLM. Agents in this category augment generation with external memory. The AI CUDA Engineer [118] leverages a vector database of high-quality kernel examples to ground the LLM's generation, ensuring syntactic correctness and adherence to best practices in low-level programming. Beyond merely reducing hallucinations regarding domain-specific syntax, this external memory mechanism serves a critical role in bridging the knowledge gap caused by the scarcity of HPC code in general pre-training corpora. KernelEvolve [119] further advances the external knowledge management paradigm by integrating a sophisticated knowledge base specifically tailored for heterogeneous AI accelerators. It employs a dynamic retrieval system to access a curated repository of optimized kernel codes and hardware-specific instructions, ensuring that the LLM can generate highly efficient kernels that are well-suited for diverse accelerator architectures.

Beyond retrieving unstructured textual context, recent work has explored utilizing structured representations as external memory to guide model inference. Work such as ReGraphT [120] 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, and the Small Language Model (SLM) will query this external memory using monte carlo graph search to retrieve optimal optimization trajectories and context-aware few-shot prompts. With the help of this structured external memory, the SLM achieves LLM-level performance while circumvents the privacy risks and excessive computing overhead.

4.3 Hardware Profiling Integration

The third dimension addresses the "blindness" of standard LLMs to hardware states. High-performance engineering requires aligning software logic with hardware topology. Agents in this category leverage "profiling" in a dual sense: they configure the agent's persona profile with hardware specifications, while iteratively reasoning over performance profiling feedback.

CUDA-LLM [121] incorporates detailed target GPU specifications, including the number of streaming multiprocessors, available shared memory, warp size, and compute capabilities—into the agent's prompt. Simultaneously, the framework aggregates granular compilation logs and comprehensive performance metrics from the runtime environment to guide the optimization process. Taking granularity further, PRAGMA [122] establishes a closed feedback loop using a specialized profiling module. By parsing low-level metrics from tools like Nsight Compute and Linux perf across heterogeneous CPU and GPU platforms, PRAGMA's conducting module translates quantitative data into interpretable natural language suggestions, enabling the LLM to reason about bottlenecks rather than just trial-and-error.

SwizzlePerf [122] explicitly tackles swizzling problem, the reordering of data or workload mappings to align with hardware topology and enhance spatial-temporal locality. The approach explicitly injects precise architectural specifications, including the number of XCDs, cache sizes, and the block-scheduling policy into the prompt context. Besides, by restricting the search space specificaslly to swizzling patterns, it establishes a fixed optimization objective focused solely on maximizing the L2 cache hit rate, effectively decoupling the optimization process from extrinsic noise such as kernel launch overheads. This allows the model to generate hardware-aligned "swizzling" patterns that map thread blocks to data locations, achieving speedups that human experts require weeks to engineer. Moreover, recognizing the importance of incorporating hardware information, researchers further integrate six tools related to compilation, profiling and benchmarking into a cohesive framework [123]. The exposing of a compact toolset to the language model significantly enhances the overall efficiency in the development and tuning process.

4.4 Multi-Agent Orchestration

Recognizing that kernel development inherently involves heterogeneous skills ranging from algorithmic planning and low-level coding to correctness verification and performance debugging, recent works increasingly adopt multi-agent designs that explicitly decompose these responsibilities into coordinated roles.

STARK [124] exemplifies this paradigm by structuring kernel generation as a collaboration among Plan, Code, and Debug agents, closely mirroring a human engineering workflow. The planning agent formulates high-level strategies for the coding agent, while the debug agent synthesizes compiler errors and runtime feedback to iteratively repair and refine candidate kernels. This clear separation of concerns enables STARK to effectively bridge high-level intent and low-level implementation details. AKG [125] similarly demonstrates the benefits of modularizing kernel generation into specialized sub-tasks handled by distinct agents, and achieve the cross-platform kernel synthesis. Other systems emphasize iterative generation–evaluation loops between complementary agent roles. CudaForge [126] adopts a two-agent setup consisting of a coder agent and a judge agent, which repeatedly interact to generate, validate, and optimize CUDA kernels using hardware-level feedback. This lightweight division of labor achieves strong generalization across GPU architectures and base models. KernelFalcon [127] employs a multi-agent system to tackle the challenge of GPU kernel generation of full machine learning architectures. The system specifically addresses hierarchical task decomposition and delegation through coordinated manager and worker agents, moving beyond single, isolated operators to handle end-to-end model acceleration.

Beyond generating kernels from scratch, several works leverage multi-agent collaboration to optimize existing implementations. Astra [128] targets performance tuning of production kernels extracted from SGLang, orchestrating multiple specialized agents for planning, coding, testing, and profiling. By iteratively refining a seed kernel through reflective feedback, Astra demonstrates that agent specialization is particularly effective for navigating fine-grained performance trade-offs. In addition, GEAK [24] developed for AMD GPUs, integrates agents for kernel generation, evaluation, reflection, and optimization within a Triton-based workflow, achieving substantial gains on rigorously designed benchmarks. Collectively, these works highlight a common theme: by explicitly assigning distinct responsibilities to cooperating agents, multi-agent frameworks provide a structured and scalable approach to tackling the complexity of kernel generation and optimization across architectures and workloads.

In summary, the field of LLM-based kernel generation is transitioning from naive code synthesis to sophisticated Agentic AI Systems. By combining evolutionary search strategies, retrieval-augmented memory, granular hardware profiling, and multi-agent collaboration, these systems are beginning to rival human experts in producing high-performance, hardware-efficient kernels.

5 Datasets for LLM-Based Kernel Generation

The efficacy of Large Language Models (LLMs) in high-performance kernel generation relies critically on the availability of domain-specific data. Unlike general software engineering, kernel generation requires models to internalize hardware intrinsics, parallel execution semantics, and memory hierarchy constraints. We argue that high-quality data in this domain is defined not merely by volume, but by its ability to bridge the semantic gap between high-level algorithms

and low-level hardware optimizations. In this section, we survey the data landscape and organize resources into three categories: (1) *Training Corpora*, covering both structured datasets and raw kernel repositories; (2) *Knowledge Bases*, which we identify as essential for grounding RAG systems; and (3) *Construction Methodology*, discussing strategies to overcome data scarcity. A comprehensive index is provided in Appendix Table 2.

5.1 Training Corpora: Structured Datasets and Source Hierarchies

We posit that the utility of training data has evolved from unstructured scraping to targeted, structure-aware curation. We summarize training corpora by their readiness for model pipelines, contrasting pre-processed instruction datasets with a stratified ecosystem of raw repositories.

Structured Datasets. Structured datasets represent the highest-value signal for instruction tuning, as they explicitly pair intent with optimization. Several recent contributions substantially lower the barrier for cross-language supervision. The KernelBook dataset [129] supports cross-language alignment by pairing high-level PyTorch operator specifications with optimized Triton implementations at scale. KernelBench further releases the dataset kernelbench-samples, which contains kernel code snapshots generated by the evaluated model during the iterative optimization process, as well as the corresponding performance profiling data. For instruction tuning in HPC settings, HPC-Instruct [130] provides curated instruction–response pairs spanning CUDA, OpenMP, and MPI tasks, enabling models to learn parallelization logic beyond surface syntax. In addition, the HPC-focused subset of The Stack v2 [131] offers a large unsupervised corpus when filtered for CUDA- and Triton-related content, serving as a foundational baseline for pre-training.

Source Code Repositories. While structured datasets provide supervision, the vast majority of domain knowledge remains latent in open-source repositories. To capture the full kernel development lifecycle, we stratify these repositories into three levels based on their abstraction and semantic role.

- **Operator and Kernel Libraries**. The first and most fine-grained layer consists of operator-centric kernel repositories, which encode a diverse collection of hardware-aware optimization patterns. These repositories expose concrete implementations of performance-critical primitives and thus provide high-quality signals that can be distilled through data cleaning and normalization. A prominent subset targets dense linear algebra, where libraries such as CUTLASS [67] offer highly optimized matrix multiplication kernels with carefully engineered tiling, pipelining, and memory layouts. Another major category focuses on attention-related operators, which have become central to LLM workloads. Representative examples include FlashAttention [12], FlagAttention [132], AoTriton [133], and xFormers [134], each encoding distinct optimization strategies for memory-efficient attention across hardware platforms. In addition, a growing class of repositories provides LLM-oriented operator implementations that target both training and inference workloads such as Liger-Kernel [135] and FlagGems [136]. Beyond full-precision kernels, several repositories concentrate on model quantization and inference acceleration. Libraries such as Bitsandbytes [137], GemLite [138], FlashInfer [139], FBGEMM [140], and NVIDIA Transformer Engine [141] implement low-precision arithmetic, fused operators, and numerically stable mixed-precision pipelines. Collectively, these operator repositories form a rich corpus of reusable optimization patterns spanning matrix computation, attention mechanisms, and quantized inference.
- **Framework and System Integration Code**. The second layer consists of kernels embedded within deep learning frameworks and runtime systems, where operator implementations are designed to support end-to-end usability rather than standalone performance. Kernels cannot be fully understood in isolation; their efficiency and behavior depend heavily on the surrounding runtime environment, including execution graphs, memory management, and scheduling policies. Core frameworks such as TensorFlow [142], PyTorch (ATen) [143], and PaddlePaddle [144] provide extensive collections of operator implementations that balance generality, extensibility, and performance. Beyond general-purpose frameworks, inference systems such as vLLM [85], SGLang [145], llama.cpp [146], and TensorRT-LLM [147] incorporate specialized kernels tightly coupled with system-level optimizations, for example, dynamic batching and KV-cache management. Similarly, training frameworks such as DeepSpeed [148] introduce custom operators to support large-scale model parallelism and memory-efficient training. After appropriate cleaning, these framework-level kernels expose how operator implementations are adapted to practical runtime constraints and thus serve as valuable training data.
- **Domain-Specific Languages (DSLs) and Emerging Abstractions**. The third layer comprises domain-specific languages (DSLs) and their associated tutorials and reference implementations, which offer an intermediate abstraction between high-level intent and low-level kernel code. These resources often include pedagogical examples and reusable operator templates that complement large-scale repositories. DSLs such as Triton, along with emerging languages and abstractions like TileLang and cuTile, provide both language specifications and concrete kernel implementations that illustrate canonical optimization patterns. Tutorial code, sample libraries,

and reference operators written in these DSLs can be directly harvested as kernel corpora. Incorporating DSL-based resources allows models to learn kernel construction principles in a structured manner, while reducing overfitting to framework-specific or hardware-specific implementation details.

5.2 Knowledge Bases: Documentation and Heuristics

Beyond executable code, curated domain knowledge also plays a critical role in training kernel-aware models. Such knowledge can be distilled into pre-training corpora to enrich model understanding, or integrated as external knowledge bases to support retrieval-augmented generation in agent-based systems. We argue that such knowledge corpora must be grounded in authoritative specifications to minimize hallucinations. For example, the *CUDA C++ Programming Guide* [149] and *PTX ISA Reference* [150] define legal syntax, while *NVIDIA Architecture Tuning Guides* [151] summarize optimization heuristics. However, formal docs often omit engineering "tribal knowledge". Thus, community indices such as GPU-MODE [152] and Triton Index [153] are valuable for debugging, and meta-indices (e.g., Awesome-CUDA) connect isolated topics. Pedagogical resources like LeetCUDA [154] and Triton Puzzles [155] aid in step-wise reasoning. Colfax Research [156] serves as a specialized technical hub dedicated to High-Performance Computing (HPC) and AI, offering in-depth articles and tutorials on NVIDIA GPU architectures, CUDA optimization, and deep learning frameworks. Finally, profiling outputs from Nsight Compute [157] and Triton-Viz [158] serve as grounded descriptions of runtime behavior, bridging the gap between static code and dynamic performance.

5.3 Dataset Construction Methodologies

Developing robust kernel generation models requires addressing the scarcity of expert-written code. From the data sources discussed above, we distill three recurring methodologies for obtaining supervision signals, each reflecting a different trade-off between scale and quality. *Expert-curated supervision* yields the highest-quality samples but scales poorly. *Cross-implementation supervision* exploits functionally equivalent implementations from multi-backend libraries (e.g., vLLM and xFormers), enabling models to learn optimization patterns by contrasting alternative realizations of the same computation. *Compiler-derived supervision* addresses data scarcity through programmatic synthesis, where compiler toolchains such as TorchInductor generate valid kernels from arbitrary computation graphs. Across all settings, effective pipelines must enforce functional correctness via execution-based filtering, retaining only samples that match trusted reference outputs within a numerical tolerance.

6 Benchmark

This chapter focuses on the systematic benchmarking of kernel generation, and provides a structured overview of representative evaluation benchmarks, including both evaluation metrics and benchmark datasets. By jointly reviewing the design of evaluation metrics and the composition of test suites, this chapter aims to establish a clear and coherent benchmark comparison for kernel generation research, thereby laying a solid foundation for subsequent method comparison and performance analysis.

6.1 Metrics

Several factors should be considered when evaluating the performance of the operator implementation: correctness, efficiency, compatibility, etc. To build a comprehensive evaluation, existing benchmarks generally adopt execution-based unit tests, where the generated kernels will be compared with the standard implementations of CUDA/PyTorch. Given the instability of operator generation, each testing task usually involves multiple evaluations across k random samples among n times of generation.

Correctness primarily includes two aspects based on difficulty: (1) successful compilation and (2) consistency with the reference in multiple input-output comparisons. The compilation pass rate, as a prerequisite for the testing sample pass rate, evaluates the probability that the implementations can run without error. The testing sample pass rate measures whether the generated implementation aligns with the reference in terms of input-output behavior. Among various metrics used in code generation, *pass@k* is widely chosen, which calculates the probability that at least one correct implementation is generated in k trials. The standard estimator is defined as:

$$\text{pass@}k \triangleq \mathbb{E}\left[1 - \binom{n-c}{k} \Big/ \binom{n}{k}\right], \tag{1}$$

where the expectation is taken over kernel tasks and prompts, c is the number of correct kernel implementations.

Efficiency is another principal goal that kernel evaluation focus on. Typical metrics on efficiency include speed-up and resource utilization. Speedup@k measures how much faster a generated implementation is compared with the baselines

Table 1: Benchmark datasets for code generation and optimization in kernel generation. Metrics: Ⓒ Correctness, Ⓢ Speedup, Ⓔ Efficiency, ⓕ $fast_p$, Ⓟ Perf, Ⓢ Similarity. Hardware Platforms: Ⓝ NVIDIA GPUs, Ⓗ HUAWEI NPUs, Ⓖ Google TPUs, Ⓐ AMD GPUs.

Name	Time	Metrics	Hardware	Description
ParEval [159]	2024.01	ⒸⓈⒺ	ⓃⒶ	420 expert-selected tasks across 12 algorithmic domains for benchmarking general parallel code generation.
KernelBench[111]	2025.02	Ⓒⓕ	Ⓝ	250 PyTorch-to-CUDA kernel generation tasks, curated from popular GitHub repositories and official PyTorch operators, for evaluating AI/DL kernel generation.
TritonBench[160]	2025.02	ⒸⓈ Ⓢ Ⓔ*	Ⓝ	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).
MultiKernelBench[161]	2025.07	ⒸⓈ	ⒽⓃⒼ	285-task benchmark across 14 operator categories for multi-platform DL kernel synthesis.
TritonBench-revised & ROCm Triton Benchmark[24]	2025.07	ⒸⓈ	Ⓐ	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[114]	2025.09	ⒸⓈ	Ⓝ	A robustness-focused benchmark featuring 9 specialized deep learning task categories, derived by refining and extending KernelBench.
CUDAEval[120]	2025.10	ⒸⓈ	Ⓝ	Leveraging 313 curated tasks from the the Stack v2 to benchmark the efficacy of reasoning transfer in CUDA code optimization.
TritonGym[162]	2025.10	ⒸⓅ	Ⓝ	Focus on derivative and synthetic tasks by mutating operator semantics to create Out-of-Distribution (OOD) kernels and introducing domain-specific language extension tasks that are rarely encountered in standard datasets.

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

by calculating

$$\text{speedup@}k \triangleq \mathbb{E}\left[\sum_{j=1}^{n}\left(\binom{j-1}{k-1}T^{\text{base}}\right) / \left(\binom{n}{k}T_j\right)\right], \tag{2}$$

where T_j is the running time of the j-th generated implementation while T^{base} is the time consumed by the baseline. Note that the implementations are sorted by their performance, i.e., T_1 corresponds to the slowest and T_n to the fastest. Efficiency@k refers to how effectively the generated operators utilize computation resources (such as kernel threads etc.) during execution as discussed in ParEval [159].

$$\text{efficiency@}k \triangleq \mathbb{E}\left[\sum_{j=1}^{n}\left(\binom{j-1}{k-1}T^{\text{base}}\right) / \left(\binom{n}{k}mT_{j,m}\right)\right], \tag{3}$$

where m is the number of computation resources and $T_{j,m}$ is the time consumed by the j-th generated operators when running with m computation resources.

Compatibility should also be considered when evaluating operator generation techniques. Specifically, since operator generation relies on different hardware platforms and involves multiple low-source programming languages, LLMs tend to exhibit significant performance variation. For example, LLM is a good at writing GPU operator but fails in generating correct operators when facing to NPU tasks. To thoroughly assess robustness, many benchmarks strive to create more diverse testing tasks and testing environments, detailed in the following subsections.

In addition, some combined metrics are also used to evaluate multiple aspects of performance as the same time. For example, *Perf*@K measures how close the best result from K generated kernels is to a human expert performance. The $fast_p$ jointly evaluates the functional correctness and runtime performance of generated kernels. *Similarity*

(CODEBLEU[163]) used 4 items (n-gram, weighted n-gram, syntax and dataflow) to measure the similarity between the generated code and the reference code.

6.2 Benchmark Datesets

A reliable benchmark depends not only on well-designed evaluation metrics, but also, and often more critically, on the representativeness, diversity, and controllability of its test data. As summarized in Table 1, we systematically review the major benchmark suites used for evaluating automated kernel generation. Collectively, these benchmarks exhibit consistent trends, including a shift from basic to composite metrics, from single-GPU to multi-platform hardware support, and from general parallel computation to derived, synthetic, and rare operator evaluation.

• **Key metrics** Regarding evaluation metrics, benchmark design evolves from basic correctness-oriented evaluation toward composite metrics that capture multiple optimization objectives. Early benchmarks primarily rely on functional correctness and raw speedup, as exemplified by ParEval and KernelBench. In contrast, more recent benchmarks adopt richer metric sets, including efficiency and similarity in TritonBench, $Perf@K$ metrics in TritonGym, and robustness-oriented speedup evaluation in Robust-kbench. This progression reflects the inherently multi-objective nature of kernel generation and optimization.

• **Hardware platforms** From the hardware platform perspective, benchmarks evolve from being tightly coupled to a single GPU vendor toward increasingly comprehensive multi-platform coverage. As shown in Table 1, early benchmarks such as ParEval, KernelBench, and TritonBench exclusively target NVIDIA GPUs. Later efforts, exemplified by MultiKernelBench, explicitly incorporate heterogeneous accelerators spanning NVIDIA GPUs, HUAWEI NPUs, and Google TPUs, while ROCm triton benchmark and TritonBench-revised further extend evaluation to AMD GPUs.

• **Evaluation content** In terms of evaluation content, benchmarks progressively move from general parallel computation and publicly available kernels toward derivative, synthetic, and rare kernel generation tasks. Specifically, ParEval focuses on 420 expert-selected parallel tasks across 12 algorithmic domains, while KernelBench and TritonBench emphasize real-world PyTorch-to-CUDA or Triton kernel generation curated from popular GitHub repositories and The Stack v2. More recent benchmarks such as TritonGym further expand the scope by introducing mutated operator semantics, out-of-distribution kernels, and DSL extension tasks that are rarely encountered in standard datasets to evaluate the generalization ability.

7 Challenges and Opportunities

While the integration of large language models (LLMs) and agentic workflows has shown strong potential for automating kernel generation, the field remains at an early stage of development. Bridging the gap between promising prototypes and production-grade systems requires addressing a set of interrelated challenges, including data scarcity, limitations in agentic reasoning and engineering discipline, the lack of scalable synthesis and training infrastructure, and insufficiently robust evaluation practices. This section examines these challenges and highlights emerging research directions spanning data, agents, infrastructure, evaluation, and human–AI collaboration, which are likely to shape the next generation of AI-driven kernel generation and optimization systems.

7.1 Data Scarcity and Synthetic Scaling

Despite growing interest in LLM-driven kernel generation, progress toward production-grade performance remains fundamentally constrained by data scarcity. High-performance kernels exhibit a pronounced long-tail distribution and are sparsely represented in existing code corpora, thereby limiting effective scaling and generalization. This scarcity manifests along two critical dimensions. First, most available datasets still lack deep, hardware-aware domain knowledge, including architectural constraints, memory hierarchy considerations, and performance-critical design patterns. Second, most existing corpora predominantly capture only final optimized kernels, while omitting optimization trajectories—such as intermediate attempts, failure cases, and performance-driven refinements—that encode essential procedural knowledge. Addressing these limitations requires elevating data availability to a first-class research bottleneck. Promising directions include systematic kernel dataset construction, large-scale synthetic data generation, and the collection of execution-driven optimization processes. Such data can support a wide range of learning paradigms, including pretraining, supervised fine-tuning, and reinforcement learning, and may be crucial for enabling meaningful scaling behavior in kernel generation systems.

7.2 Advancing Agentic Reasoning and Engineering Standards

Current agent-based approaches to kernel optimization largely adopt predefined, workflow-driven paradigms, in which agents operate through reactive trial-and-error loops. While such designs enable limited automation, they often lack

the autonomy required to sustain long-horizon optimization objectives. As a result, agents frequently degenerate into repetitive or unproductive exploration, ultimately exhausting context windows without converging to meaningful improvements.

Therefore, realizing an advanced kernel agent requires overcoming several challenges. First, agent autonomy should be enhanced by reducing reliance on rigid, hand-crafted workflows in favor of self-directed planning and iterative reflection, enabling agents to decompose tasks, revise strategies, and manage memory dynamically. Second, to support principled reasoning rather than ad-hoc pattern matching, external domain knowledge dispersed across documentation and expert heuristics must be systematically integrated into structured knowledge bases. Finally, advancing toward production-grade systems demands rigorous engineering standards, including formal specifications, verification protocols, and disciplined development practices, to ensure reliability, safety, and reproducibility alongside performance gains. Collectively, addressing these challenges is critical for transforming agentic kernel optimization from exploratory automation into a robust, engineering-grade capability

7.3 Scalable Infrastructure for Synthesis and Training

Scalable infrastructure is a foundational yet often underemphasized bottleneck in AI-driven kernel synthesis and optimization. Whether for large-scale data sampling, synthetic data generation, or reinforcement learning, system efficiency and experimental fidelity are tightly coupled to the underlying execution and training infrastructure. A key challenge arises from the severe latency mismatch between model inference and kernel verification. Unlike natural language tasks, kernel validation requires a costly compilation toolchain and hardware-level execution, resulting in low-throughput feedback loops that fundamentally limit scalable exploration and learning. This bottleneck constrains not only reinforcement learning but also the collection of diverse, execution-grounded synthetic data.

Addressing this challenge calls for infrastructure that cleanly decouples model reasoning from environment execution. Promising directions include standardized, gym-like kernel optimization environments that abstract interactions among agents, compilers, and profilers, while supporting distributed and asynchronous execution at scale. Such infrastructure can enable massive parallel sampling, reduce engineering overhead, and expose rich, multi-objective feedback signals—including latency, correctness, and resource utilization—necessary for robust training. Ultimately, advances in scalable infrastructure are critical for transforming kernel synthesis from low-throughput experimentation into a systematic, data-driven learning process.

7.4 Evaluation Robustness and Cross-Ecosystem Generalization

A key open challenge in AI-driven kernel generation is the lack of robust and comprehensive evaluation. Many existing benchmarks assess performance under narrowly defined settings, limiting confidence in reported gains and obscuring generalization behavior.

At the operator level, evaluations are often restricted to fixed input shapes, limited precision regimes, and a small set of forward-pass primitives. Such protocols fail to reflect real-world workloads, where kernels must generalize across varying shapes, data types, and execution contexts. Moreover, backward operators, fused kernels, and composite workloads remain underrepresented, raising concerns about the applicability of current methods beyond canonical benchmarks. Evaluation limitations are further compounded by ecosystem concentration. Most approaches are developed and validated within the NVIDIA ecosystem, predominantly targeting CUDA-based kernels. This focus leaves open the question of whether current methodological paradigms such as data synthesis, agentic optimization, and reinforcement learning—can be instantiated across diverse hardware platforms, programming models, and toolchains. Addressing these gaps requires evaluation protocols that jointly assess robustness and generalization across shapes, operators, and ecosystems, providing a more reliable foundation for measuring progress in kernel synthesis research.

7.5 Human-AI Collaboration for Kernel Generation

Beyond fully automated approaches, human–AI collaboration represents an important and complementary paradigm for kernel generation. An open research question is how to systematically combine agentic exploration with human expertise to expand the design space and improve controllability in performance-critical settings.

A central challenge in this paradigm is explainability. To enable effective collaboration and oversight, agents must provide interpretable rationales for their optimization decisions such as tiling strategies and memory layouts, which allow humans to reason about, validate, and refine system behavior. Without transparent explanations, human involvement is reduced to ad-hoc intervention, limiting the effectiveness of collaborative workflows. Equally important is the design of interaction paradigms that support principled division of labor between humans and agents. Future systems should facilitate mixed-initiative collaboration, in which humans specify high-level intent, constraints, or optimization

objectives, while agents perform implementation, exploration, and parameter tuning. Establishing such interaction models is essential for enabling flexible and scalable collaboration without sacrificing automation benefits.

8 Conclusion

This survey highlights the transformative potential of large language models and agentic workflows for automating high-performance kernel generation, signaling a paradigm shift from static code synthesis toward dynamic, feedback-driven optimization. By synthesizing recent advances in supervised fine-tuning, reinforcement learning, and multi-agent orchestration, together with progress in kernel-centric dataset curation and benchmark development, we present a unified perspective on this rapidly evolving research area. Looking ahead, future work must move beyond rigid workflow-based designs toward deeper agentic reasoning and self-evolving optimization capabilities, while simultaneously addressing ecosystem bias to support diverse hardware platforms and broadly generalizable kernel-generation methodologies. Ultimately, addressing these challenges is critical not only for reducing the burden of manual kernel engineering and freeing human experts from low-level optimization complexity, but also for systematically improving software efficiency and unlocking productivity gains in the era of rapidly scaling AI infrastructure.

Acknowledgments

The authors would like to thank Jinjin Tian and Zhen Wang for their valuable expertise and careful review of the kernel and operator-related aspects of this survey. Additionally, we extend our appreciation to Liangdong Wang, Jiabei Chen, and Yao Xu for their insights on the LLM4Kernel content, as well as to Hao Liang, Tianyu Guo and Runming He for their helpful suggestions on the data section. Finally, we thank Jian Tao, Dongxu Han, and Feng Liao for their advice on the Benchmark analysis.

References

[1] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[2] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

[3] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

[4] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

[5] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[6] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[7] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. Nvidia a100 tensor core gpu: Performance and innovation. *IEEE Micro*, 41(2):29–35, 2021.

[8] Jack Choquette. NVIDIA Hopper H100 GPU: Scaling performance. *IEEE Micro*, 43(3):9–17, 2023.

[9] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, et al. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *Proceedings of the 50th annual international symposium on computer architecture*, pages 1–14, 2023.

[10] Heng Liao, Jiajin Tu, Jing Xia, Hu Liu, Xiping Zhou, Honghui Yuan, and Yuxing Hu. Ascend: a scalable and unified architecture for ubiquitous deep neural network computing: Industry track paper. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 789–801. IEEE, 2021.

[11] Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, et al. Rwkv: Reinventing rnns for the transformer era. *arXiv preprint arXiv:2305.13048*, 2023.

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

[13] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

[14] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *Advances in Neural Information Processing Systems*, 37:68658–68685, 2024.

[15] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for? *Queue*, 6(2):40–53, 2008.

[16] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th annual international symposium on Computer architecture*, pages 152–163, 2009.

[17] Vasily Volkov. Better performance at lower occupancy. In *Proceedings of the GPU technology conference, GTC*, volume 10, page 16. San Jose, CA, 2010.

[18] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[19] Peng Wu. Pytorch 2.0: The journey to bringing compiler technologies to the core of pytorch (keynote). In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, pages 1–1, 2023.

[20] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[21] Philippe Tillet, Hsiang-Tsung 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*, pages 10–19, 2019.

[22] Zacharias V. Fisches, Sahan Paliskara, Simon Guo, Alex Zhang, Joe Spisak, Chris Cummins, Hugh Leather, Gabriel Synnaeve, Joe Isaacson, Aram Markosyan, and Mark Saroufim. Kernelllm: Making kernel development more accessible, 6 2025. Corresponding authors: Aram Markosyan, Mark Saroufim.

[23] Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. Kevin: Multi-turn rl for generating cuda kernels. *arXiv preprint arXiv:2507.11948*, 2025.

[24] Jianghui Wang, Vinay Joshi, Saptarshi Majumder, Chao Xu, Bin Ding, Ziqiong Liu, Pratik Prabhanjan Brahma, Dong Li, Zicheng Liu, and Emad Barsoum. Geak: Introducing triton kernel ai agent & evaluation benchmarks. *arXiv preprint arXiv:2507.23194*, 2025.

[25] Anjiang Wei, Tianran Sun, Yogesh Seenichamy, Hang Song, Anne Ouyang, Azalia Mirhoseini, Ke Wang, and Alex Aiken. Astra: A multi-agent system for gpu kernel performance optimization. *arXiv preprint arXiv:2509.07506*, 2025.

[26] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[27] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

[28] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[29] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 30, 2017.

[30] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[31] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[32] Ahmed El-Kishky, Alexander Wei, et al. Competitive programming with large reasoning models. *arXiv preprint arXiv:2502.06807*, 2025.

[33] Daniel Kahneman. Thinking, fast and slow. *Farrar, Straus and Giroux*, 2011.

[34] OpenAI. Introducing GPT-5. https://openai.com/index/introducing-gpt-5/, 2025. Official Announcement.

[35] Aixin Liu, Bei Feng, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[36] Anthropic. Introducing Claude 4. https://www.anthropic.com/news/claude-4, 2025. Official Announcement.

[37] Google DeepMind. Introducing Gemini 2.0: our new AI model for the agentic era. https://blog.google/technology/google-deepmind/google-gemini-ai-update-december-2024/, 2024. Official Blog Post.

[38] Meta AI. The Llama 4 herd: The beginning of a new era of natively multimodal AI innovation. https://ai.meta.com/blog/llama-4-multimodal-intelligence/, 2025. Official Blog Post.

[39] Binyuan Hui, Jian Yang, et al. Qwen2.5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

[40] Mohamed Amine Ferrag, Norbert Tihanyi, and Merouane Debbah. From llm reasoning to autonomous ai agents: A comprehensive review. *arXiv preprint arXiv:2504.19678*, 2025.

[41] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, 2024.

[42] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022.

[43] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551, 2023.

[44] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.

[45] Zeyu Zhang, Quanyu Dai, Xiaohe Bo, et al. A survey on the memory mechanism of large language model-based agents. *ACM Transactions on Information Systems*, 43(6):1–47, 2025.

[46] Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pages 1–22, 2023.

[47] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*, 3(4), 2023.

[48] Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in Neural Information Processing Systems*, 36:51991–52008, 2023.

[49] Kunlun Zhu, Hongyi Du, Zhaochen Hong, Xiaocheng Yang, Shuyi Guo, Daisy Zhe Wang, Zhenhailong Wang, Cheng Qian, Robert Tang, Heng Ji, et al. Multiagentbench: Evaluating the collaboration and competition of llm agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8580–8622, 2025.

[50] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of large language model based agents: A survey. *Science China Information Sciences*, 68(2):121101, 2025.

[51] Zhizheng Wang, Qiao Jin, Chih-Hsuan Wei, Shubo Tian, Po-Ting Lai, Qingqing Zhu, Chi-Ping Day, Christina Ross, Robert Leaman, and Zhiyong Lu. Geneagent: self-verification language agent for gene-set analysis using domain databases. *Nature Methods*, pages 1–9, 2025.

[52] Anthropic. Developing a computer use model. https://www.anthropic.com/news/developing-computer-use, 2024. Introduced the capability for LLMs to control GUIs and terminals.

[53] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024.

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

[55] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

[56] William R Mark, R Steven Glanville, Kurt Akeley, and Mark J Kilgard. Cg: A system for programming graphics hardware in a c-like language. In *ACM SIGGRAPH 2003 Papers*, pages 896–907. 2003.

[57] Randima Fernando and Mark J Kilgard. *The Cg Tutorial: The definitive guide to programmable real-time graphics*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[58] Ian Buck, Theresa Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM transactions on graphics (TOG)*, 23(3):777–786, 2004.

[59] NVIDIA Corporation. *GeForce 8800 GTX/GTS Technical Brief*, 2006. Official technical brief.

[60] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[61] NVIDIA Corporation. *CUDA Programming Guide, Version 1.0*, 2007. Official technical document.

[62] David Luebke, Mark Harris, Naga Govindaraju, Aaron Lefohn, Mike Houston, John Owens, Mark Segal, Matthew Papakipos, and Ian Buck. Gpgpu: general-purpose computation on graphics hardware. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, pages 208–es, 2006.

[63] David Luebke. Cuda: Scalable parallel programming for high-performance scientific computing. In *2008 5th IEEE international symposium on biomedical imaging: from nano to macro*, pages 836–838. IEEE, 2008.

[64] NVIDIA Corporation. *cuBLAS Library User Guide, Version 2.0*, 2008. Official technical document.

[65] NVIDIA Corporation. *Thrust Library Guide, Version 1.7*, 2013. Official technical document.

[66] 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.

[67] Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. CUTLASS: CUDA templates for linear algebra subroutines, 2017. Version 3.x, accessed 2025.

[68] NVIDIA Corporation. *CUDA Programming Guide, Version 9.0*, 2017. Section B.7, official technical documentation.

[69] Vasily Volkov. *Understanding latency hiding on GPUs*. University of California, Berkeley, 2016.

[70] NVIDIA Corporation. *CUDA C++ Best Practices Guide, Version 12.0*, 2021. Chapters 7–9, official technical documentation.

[71] Anne C Elster and Tor A Haugdahl. Nvidia hopper gpu and grace cpu highlights. *Computing in Science & Engineering*, 24(2):95–100, 2022.

[72] Weile Luo, Ruibo Fan, Zeyu Li, Dayou Du, Hongyuan Liu, Qiang Wang, and Xiaowen Chu. Dissecting the nvidia hopper architecture through microbenchmarking and multiple level analysis. *arXiv preprint arXiv:2501.12084*, 2025.

[73] NVIDIA Corporation. NVIDIA Blackwell Architecture Technical Overview. https://resources.nvidia.com/en-us-blackwell-architecture, 2025. Accessed: 2025.

[74] John L Hennessy and David A Patterson. A new golden age for computer architecture. *Communications of the ACM*, 62(2):48–60, 2019.

[75] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. Survey of machine learning accelerators. In *2020 IEEE high performance extreme computing conference (HPEC)*, pages 1–12. IEEE, 2020.

[76] AMD Corporation. AMD Instinct MI325X Accelerators. https://www.amd.com/en/products/accelerators/instinct/mi300/mi325x.html, 2025. Accessed: 2025.

[77] Intel Corporation. *oneAPI Programming Guide*, 2020. Official documentation.

[78] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM SIGARCH Computer Architecture News*, 42(1):269–284, 2014.

[79] Moore Threads. Moore threads: Full-stack gpu computing platform. https://en.mthreads.com/, 2025. Accessed: 2025.

[80] Iluvatar CoreX. Tiangai 100 (BI-V100): High-performance general-purpose GPGPU technical whitepaper. https://www.iluvatar.com/, 2021. Accessed: 2025.

[81] Simon J Pennycook, Jason D Sewall, and Victor W Lee. Implications of a metric for performance portability. *Future Generation Computer Systems*, 92:947–958, 2019.

[82] Jingyue Wu, Artem Belevich, Eli Bendersky, Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Roune, Rob Springer, Xuetian Weng, and Robert Hundt. gpucc: an open-source gpgpu compiler. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 105–116, 2016.

[83] Triton Language Team. Triton compilation and caching. Triton GitHub Repository, 2023. Accessed: 2025-11-26.

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

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

[86] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, et al. Jax: composable transformations of python+ numpy programs. 2018.

[87] JAX Team. Pallas: A JAX kernel language. https://docs.jax.dev/en/latest/pallas/index.html, 2023. Official JAX documentation.

[88] PyTorch Team. Helion: A high-level DSL for performant and portable ML kernels. https://github.com/pytorch/helion, 2025.

[89] Meta. TLX: Triton low-level language extensions. https://github.com/facebookexperimental/triton/tree/tlx, 2025. GitHub repository, branch tlx.

[90] Triton Team. Gluon dialect. https://triton-lang.org/main/dialects/GluonDialect.html, 2025. Triton documentation.

[91] Lei Feng et al. TileLang: A composable tiled programming model for AI systems. *arXiv preprint arXiv:2504.17577*, 2025.

[92] NVIDIA Corporation. Cuda c++ standard library: Tiling primitives. https://developer.nvidia.com/cuda/tile, 2025. Accessed: 2025.

[93] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024.

[94] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[95] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

[96] Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. Efficient training of language models to fill in the middle. *arXiv preprint arXiv:2207.14255*, 2022.

[97] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

[98] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

[99] Mark Chen, Jerry Tworek, Heewoo Jun, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. The Codex paper.

[100] Cursor Team. Cursor: The ai code editor. https://github.com/cursor/cursor, 2025. Accessed: 2025.

[101] Trae Research Team, Pengfei Gao, Zhao Tian, Xiangxin Meng, Xinchen Wang, Ruida Hu, Yuanan Xiao, Yizhou Liu, Zhao Zhang, Junjie Chen, Cuiyun Gao, Yun Lin, Yingfei Xiong, Chao Peng, and Xia Liu. Trae agent: An llm-based agent for software engineering with test-time scaling. 2025.

[102] Anthropic. Claude code: An agentic coding tool that lives in your terminal. https://github.com/anthropics/claude-code, 2025. Official command line interface.

[103] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. End-to-end deep learning of optimization heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 219–232. IEEE, 2017.

[104] Chris Cummins, Zacharias V Fisches, Tal Ben-Nun, Torsten Hoefler, Michael FP O'Boyle, and Hugh Leather. Programl: A graph-based program representation for data flow analysis and compiler optimizations. In *International Conference on Machine Learning*, pages 2244–2253. PMLR, 2021.

[105] Lingcheng Kong, Jiateng Wei, Hanzhang Shen, and Huan Wang. Concur: Conciseness makes state-of-the-art kernel generation. *CoRR*, abs/2510.07356, 2025.

[106] Anonymous. Mastering sparse cuda generation through pretrained models and deep reinforcement learning. OpenReview (ICLR 2026 submission), 2026. Under double-blind review.

[107] Shangzhan Li, Zefan Wang, Ye He, Yuxuan Li, Qi Shi, Jianling Li, Yonggang Hu, Wanxiang Che, Xu Han, Zhiyuan Liu, et al. Autotriton: Automatic triton programming with reinforcement learning in llms. *arXiv preprint arXiv:2507.05687*, 2025.

[108] Jiin Woo, Shaowei Zhu, Allen Nie, Zhen Jia, Yida Wang, and Youngsuk Park. Tritonrl: Training llms to think and code triton without cheating. *arXiv preprint arXiv:2510.17891*, 2025.

[109] Xiaoya Li, Xiaofei Sun, Albert Wang, Jiwei Li, and Chris Shum. Cuda-l1: Improving cuda optimization via contrastive reinforcement learning. *arXiv preprint arXiv:2507.14111*, 2025.

[110] Songqiao Su, Xiaofei Sun, Xiaoya Li, Albert Wang, Jiwei Li, and Chris Shum. Cuda-l2: Surpassing cublas performance for matrix multiplication through reinforcement learning, 2025.

[111] Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Re, and Azalia Mirhoseini. Kernelbench: Can LLMs write efficient GPU kernels? In *Forty-second International Conference on Machine Learning*, 2025.

[112] Terry Chen, Bing Xu, and Kirthi Devleker. Automating gpu kernel generation with deepseek-r1 and inference time scaling. NVIDIA Developer Blog, 2025.

[113] BAAI. KernelGen, 2025. Accessed on: Dec. 1, 2025.

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

[115] Annan Li, Chufan Wu, Zengle Ge, Yee Hin Chong, Zhinan Hou, Lizhe Cao, Cheng Ju, Jianmin Wu, Huaiming Li, Haobo Zhang, et al. The fm agent. *arXiv preprint arXiv:2510.26144*, 2025.

[116] Ping Guo, Chenyu Zhu, Siyuan Chen, Fei Liu, Xi Lin, Zhichao Lu, and Qingfu Zhang. Evoengineer: Mastering automated cuda kernel code evolution with large language models. *arXiv preprint arXiv:2510.03760*, 2025.

[117] Martin Andrews and Sam Witteveen. Gpu kernel scientist: An llm-driven framework for iterative kernel optimization. *arXiv preprint arXiv:2506.20807*, 2025.

[118] Robert Tjarko Lange, Aaditya Prasad, Qi Sun, Maxence Faldor, Yujin Tang, and David Ha. The ai cuda engineer: Agentic cuda kernel discovery, optimization and composition. Technical report, Sakana AI, 2025.

[119] Gang Liao, Hongsen Qin, Ying Wang, Alicia Golden, Michael Kuchnik, Yavuz Yetim, Jia Jiunn Ang, Chunli Fu, Yihan He, Samuel Hsia, Zewei Jiang, Dianshi Li, Uladzimir Pashkevich, Varna Puvvada, Feng Shi, Matt Steiner, Ruichao Xiao, Nathan Yan, Xiayu Yu, Zhou Fang, Abdul Zainul-Abedin, Ketan Singh, Hongtao Yu, Wenyuan Chi, Barney Huang, Sean Zhang, Noah Weller, Zach Marine, Wyatt Cook, Carole-Jean Wu, and Gaoxiang Liu. Kernelevolve: Scaling agentic kernel coding for heterogeneous ai accelerators at meta, 2025.

[120] Junfeng Gong, Zhiyi Wei, Junying Chen, Cheng Liu, and Huawei Li. From large to small: Transferring cuda optimization expertise via reasoning graph. *arXiv preprint arXiv:2510.19873*, 2025.

[121] Wentao Chen, Jiace Zhu, Qi Fan, Yehan Ma, and An Zou. Cuda-llm: Llms can write efficient cuda kernels. *arXiv preprint arXiv:2506.09092*, 2025.

[122] Kelun Lei, Hailong Yang, Huaitao Zhang, Xin You, Kaige Zhang, Zhongzhi Luan, Yi Liu, and Depei Qian. Pragma: A profiling-reasoned multi-agent framework for automatic kernel optimization. *arXiv preprint arXiv:2511.06345*, 2025.

[123] Daniel Nichols, Konstantinos Parasyris, Charles Jekel, Abhinav Bhatele, and Harshitha Menon. Integrating performance tools in model reasoning for gpu kernel optimization. *arXiv preprint arXiv:2510.17158*, 2025.

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

[125] Jinye Du, Quan Yuan, Zuyao Zhang, Yanzhi Yi, Jiahui Hu, Wangyi Chen, Yiyang Zhu, Qishui Zheng, Wenxiang Zou, Xiangyu Chang, Zuohe Zheng, Zichun Ye, Chao Liu, Shanni Li, Renwei Zhang, Yiping Deng, Xinwei Hu, Xuefeng Jin, and Jie Zhao. Akg kernel agent: A multi-agent framework for cross-platform kernel synthesis, 2025.

[126] Zijian Zhang, Rong Wang, Shiyang Li, Yuebo Luo, Mingyi Hong, and Caiwen Ding. Cudaforge: An agent framework with hardware feedback for cuda kernel optimization. *arXiv preprint arXiv:2511.01884*, 2025. https://arxiv.org/abs/2511.01884.

[127] PyTorch Team and Contributors. Kernelfalcon: Autonomous GPU kernel generation via deep agents. https://pytorch.org/blog/kernelfalcon-autonomous-gpu-kernel-generation-via-deep-agents/, 2024. Accessed: 2026-01-02.

[128] Anjiang Wei, Tianran Sun, Yogesh Seenichamy, Hang Song, Anne Ouyang, Azalia Mirhoseini, Ke Wang, and Alex Aiken. Astra: A multi-agent system for gpu kernel performance optimization. *arXiv preprint arXiv:2509.07506*, 2025.

[129] GPU MODE Team. Kernelbook: A curated collection of pytorch–triton pairs. https://huggingface.co/datasets/GPUMODE/KernelBook, 2025. Accessed 2025-12-08.

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

[131] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykcz, et al. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*, 2024.

[132] FlagOpen Team. Flagattention: A collection of memory efficient attention operators implemented in the triton language. https://github.com/flagos-ai/FlagAttention, 2023. Accessed: 2025-12-30.

[133] AMD. Aotriton: Pre-compiled triton kernels for rocm. https://github.com/ROCm/aotriton, 2024.

[134] Benjamin Lefaudeux, Francisco Massa, Diana Liskovich, Wenhan Xiong, Vittorio Caggiano, Sean Naren, Min Xu, Jieru Hu, Marta Tintore, Susan Zhang, Patrick Labatut, Daniel Haziza, Luca Wehrstedt, Jeremy Reizenstein, and Grigory Sizov. xformers: A modular and hackable transformer modelling library. https://github.com/facebookresearch/xformers, 2022.

[135] LinkedIn. Liger-kernel: Efficient triton kernels for llm training. https://github.com/linkedin/Liger-Kernel, 2024.

[136] BAAI FlagOpen team. Flagopen/flaggems: Flaggems is an operator library for large language models implemented in the triton language., 2024.

[137] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm.int8(): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339*, 2022.

[138] Dropbox, Inc. Gemlite: A lightweight machine learning framework for efficient model serving. GitHub repository.

[139] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. Flashinfer: Efficient and customizable attention engine for llm inference serving. *arXiv preprint arXiv:2501.01005*, 2025.

[140] Daya Khudia, Jianyu Huang, Protonu Basu, et al. Fbgemm: High performance fpga-like kernels for server-side inference. *arXiv preprint arXiv:2101.05615*, 2021.

[141] NVIDIA. Transformer engine: An nvidia library for accelerating transformer training with fp8. https://github.com/NVIDIA/TransformerEngine, 2022. Open-source library for FP8-based Transformer training and inference.

[142] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning, 2016.

[143] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[144] Yanjun Ma, Dianhai Yu, Tian Wu, and Haifeng Wang. Paddlepaddle: An open-source deep learning platform from industrial practice. *Frontiers of Data and Domputing*, 1(1):105–115, 2019.

[145] Lianmin Zheng, Li Li, Hao Zhang, et al. Sglang: Efficient execution of structured language model programs. *arXiv preprint arXiv:2312.07104*, 2023.

[146] Georgi Gerganov. llama.cpp: Port of facebook's llama model in c/c++. https://github.com/ggml-org/llama.cpp, 2023. Accessed: 2025-12-30.

[147] NVIDIA. Tensorrt-llm: A tensorrt toolbox for optimized large language model inference. https://github.com/NVIDIA/TensorRT-LLM, 2023.

[148] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3506, 2020.

[149] NVIDIA Corporation. *CUDA C++ Programming Guide, Version 12.4*, 2024.

[150] NVIDIA Corporation. *Parallel Thread Execution ISA Version 8.4*, 2024.

[151] NVIDIA Corporation. *NVIDIA Architecture Tuning Guides (Ampere, Hopper, Blackwell)*, 2024.

[152] GPU MODE Community. Gpu-mode community resources and lectures. https://github.com/gpu-mode/resource, 2025.

[153] Vinay Joshi et al. triton-index: A comprehensive index of triton resources. https://github.com/gpu-mode/triton-index, 2025.

[154] Yijia Shao. Leetcuda: Cuda puzzles for learning parallel programming. https://github.com/Bruce-Lee-LY/cuda_hpc, 2024.

[155] Sasha Rush. Triton puzzles. https://github.com/srush/Triton-Puzzles, 2023.

[156] Colfax Research. Colfax research: Contributing to innovations in computing. https://research.colfax-intl.com/, 2025. Accessed: 2025-12-31.

[157] NVIDIA Corporation. *NVIDIA Nsight Compute User Guide*, 2024.

[158] Deep-Learning-Profiling-Tools. Triton-viz: Visualization tools for triton kernels. https://github.com/Deep-Learning-Profiling-Tools/triton-viz, 2024. Accessed 2025-12-08.

[159] Daniel Nichols, Joshua H. Davis, Zhaojun Xie, Arjun Rajaram, and Abhinav Bhatele. Can large language models write parallel code? In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '24, page 281–294, New York, NY, USA, 2024. Association for Computing Machinery.

[160] Jianling Li, ShangZhan Li, Zhenye Gao, Qi Shi, Yuxuan Li, Zefan Wang, Jiacheng Huang, Haojie Wang, Jianrong Wang, Xu Han, Zhiyuan Liu, and Maosong Sun. Tritonbench: Benchmarking large language model capabilities for generating triton operators. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 23053–23066, 2025.

[161] Zhongzhen Wen, Yinghui Zhang, Zhong Li, Zhongxin Liu, Linna Xie, and Tian Zhang. Multikernelbench: A multi-platform benchmark for kernel generation. *arXiv eprints, pp. arXiv–2507*, 2025.

[162] Yue Guan and Yichen Lin. Tritongym: A benchmark for agentic llm workflows in triton gpu code generation. *Under review at ICLR 2026*, 2025.

[163] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

Appendix A: Comprehensive List of HPC & System Resources

The following table provides a complete catalog of the datasets, code repositories, and educational resources referenced in this work. The dates listed in the table correspond to the initial release of each github repository. It is important to note that these libraries are under active development, with continuous updates and optimizations following their inception.

Table 2: A structured overview of HPC datasets, system-level frameworks, and knowledge bases utilized in our study. Resources are categorized to align with the training strata defined in Section 5.

Date	Resource / Artifact	Description	Access
I. Structured Datasets (Hugging Face & Benchmarks)			
02/2024	**The Stack v2**	HPC Subset (Unsupervised CUDA/Triton Corpus)	[Data]
06/2024	**HPC-Instruct**	Instruction-Response Pairs for CUDA/MPI/OpenMP	[Data]
05/2025	**KernelBook**	Torch-Triton Aligned Parallel Corpus	[Data]
02/2025	**KernelBench samples**	Optimization Tasks & Performance Traces	[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 Implemented in Triton	[Code]
02/2024	**AoTriton**	AOT-compiled Triton kernels for AMD ROCm	[Code]
11/2021	**xFormers**	Hackable and Optimized Transformer Building Blocks	[Code]
08/2024	**Liger-Kernel**	Efficient Training Kernels for LLMs	[Code]
04/2024	**FlagGems**	Triton-based Operator Library (FlagAttention)	[Code]
09/2022	**Bitsandbytes**	8-bit quantization wrappers for LLMs	[Code]
09/2024	**Gemlite**	Triton Kernels for Efficient Low-Bit Matrix Multiplication	[Code]
01/2025	**FlashInfer**	Kernel Library for Efficient LLM Serving	[Code]
05/2021	**FBGEMM**	Low-precision High-performance Matrix Multiplication	[Code]
09/2022	**Transformer Engine**	FP8 acceleration library for Transformer models	[Code]
Layer 2: Framework & System Integration			
10/2016	**PyTorch (ATen)**	Foundational tensor library for C++ and Python	[Code]
06/2023	**vLLM**	High-throughput and memory-efficient serving engine	[Code]
12/2023	**SGLang**	Structured Generation Language for LLMs	[Code]
03/2023	**llama.cpp**	Port of Facebook's LLaMA model 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]
03/2024	**ThunderKittens**	Tile primitives for CUDA	[Code]
04/2024	**TileLang**	Intermediate Language for Tile-based Optimization	[Code]
06/2024	**TT-Metal**	Bare Metal Programming on Tenstorrent	[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 (Latest Release)	[Docs]
06/2007	**PTX ISA**	PTX ISA Reference	[Docs]
05/2020	**Tuning Guides**	NVIDIA Architecture Tuning Guides (Ampere/Hopper)	[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 High-Performance Computing (HPC) and AI	[Link]
09/2018	**Nsight Compute**	Kernel Profiling Guide	[Docs]