Recent Advances in Deep Learning Kernel Optimization Using Large Language Models

Tianlin Li*†, Chenxi Yang*‡, Qitong Sun†, Xiaoyu Zhang§, Qiang Hu‡, Zhe Tang¶, Sheng Chen¶, Fei Yang¶, Aishan Liu†, Xianglong Liu†, Chao Shen‖, Yves Le Traon**, and Yang Liu§
†Beihang University, China, {tianlin001,sunqt,liuaishan,xlliu}@buaa.edu.cn
‡Tianjin University, China, {3023244212,qianghu}@tju.edu.cn
§Nanyang Technological University, Singapore, {xiaoyu.zhang,yangliu}@ntu.edu.sg
¶Zhejiang Lab, China, {tangzhe,scucs,yangf}@zhejianglab.org
‖Xian Jiaotong University, China, chaoshen@mail.xjtu.edu.cn
**University of Luxembourg, Luxembourg, yves.letraon@uni.lu

***Abstract*—The exponential growth of deep learning models, especially Large Language Models (LLMs), has dramatically increased computational demands. To meet these demands, modern deep learning systems increasingly depend on specialized hardware accelerators, such as NVIDIA GPUs. The performance of deep learning workloads ultimately depends on the efficiency of computational kernels, the fundamental operators underlying these accelerators. However, these kernels are notoriously difficult to manually generate and optimize due to complex, hardware-dependent design constraints. Recent advances in LLMs are unlocking new opportunities for automated kernel generation and optimization, offering a compelling alternative to traditional labor-intensive and expert-driven approaches. This paper presents the first comprehensive survey of deep learning kernel generation and optimization using LLMs. Moreover, we provide a systematic roadmap for improving benchmarking and generation techniques in this field.**

I. Introduction

THE rapid advancement of deep learning (DL) has dramatically intensified the demand for high-performance computing [1]. To meet this demand, specialized hardware accelerators, including but not limited to NVIDIA GPUs, Huawei NPUs, and Google TPUs, have emerged as the de facto computational backbone for large-scale workloads [2–4]. These architectures, designed with massively parallel processing in mind, now power a wide spectrum of applications, from small-scale experimental models to cutting-edge large language models (LLMs) [5–7]. This massive parallel processing capability is primarily harnessed through DL kernels—programs that operate on parallel accelerators to perform computations [8]. As the fundamental operators on these accelerators, the optimization of DL kernels is critical, directly determining the overall efficiency of the system.

Although kernels are also a type of program, their generation and optimization differ from conventional code, presenting unique and formidable challenges. The primary difficulty stems from an immense and complex optimization space defined by numerous interdependent decisions, including the configuration of thread block sizes, the intricate utilization of memory hierarchies, and the application of low-level instructions [9–11]. Furthermore, their performance is highly dependent on the specific target hardware architecture [12].

The research community has long been engaged in addressing the complex challenges of DL kernel optimization[1]. Earlier approaches rely predominantly on manual tuning [13], which requires deep domain expertise to navigate a vast and highly coupled design space and to carefully balance intricate performance trade-offs among memory hierarchies, parallel execution models, and hardware-specific constraints, making manual kernel optimization both costly and difficult to scale.

Recently, the remarkable progress of LLMs has opened new opportunities for DL kernel optimization. Their strong capabilities in language understanding and code generation motivate researchers to explore how these models can assist in generating and optimizing high-performance kernels [14]. To the best of our knowledge, more than 35 LLM-driven studies have investigated kernel optimization from diverse perspectives within a single recent year, reflecting a clear and accelerating trend toward LLM-based automation.

Despite the rapid advancement and promising capabilities of LLM-based kernel generation and optimization, there is currently no systematic and comprehensive study of this emerging research area. Existing works are largely scattered across different venues and explore diverse methodological directions. This underscores the need for a timely survey of this field. To fill this gap, this survey provides a comprehensive review and comparison of existing methods. We further analyze open challenges and outline promising future research directions to guide the continued development of this field.

A. Survey Method

The survey method adopted for preparing this paper is based on an approach widely presented in previous surveys [15]. This method includes ❶ defining the objectives of the survey, ❷ defining research questions, ❸ selecting keywords for

*Equal Contribution.

[1]For simplicity, we do not distinguish between kernel generation and optimization. Here, both terms are used interchangeably to denote the process of generating new kernels.

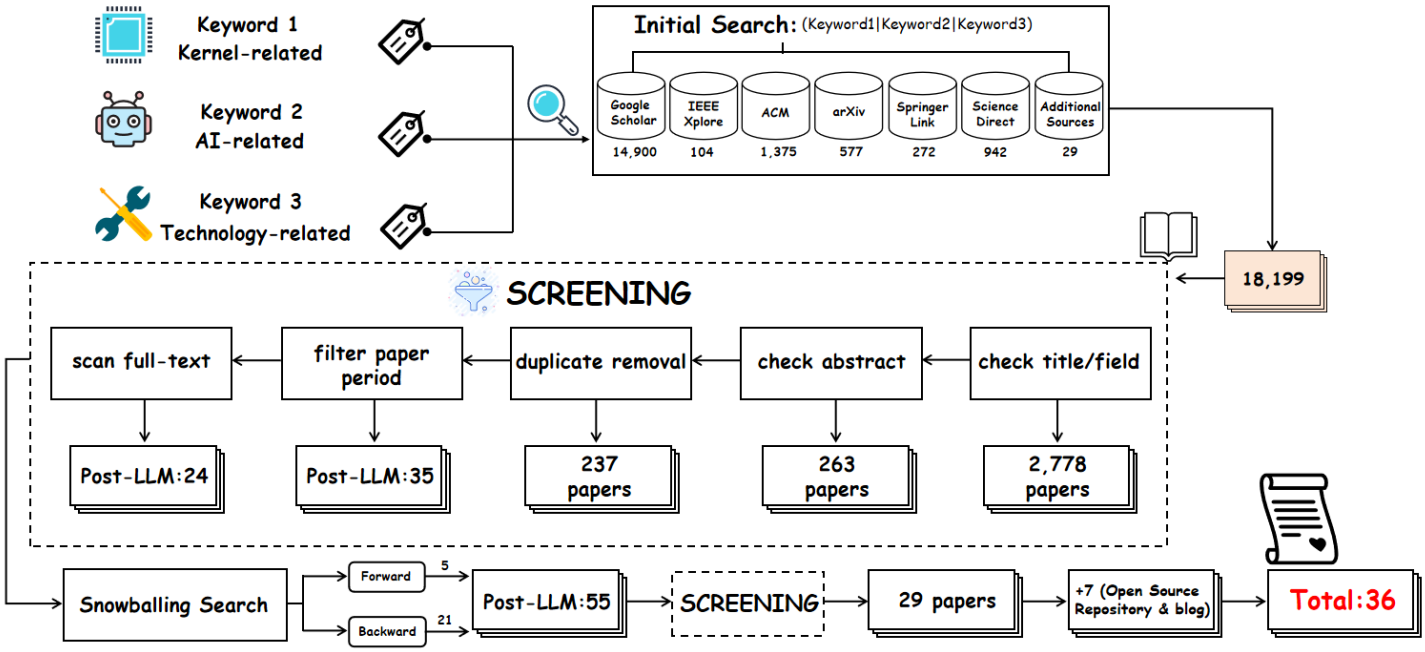

Figure 1. The workflow and methodology of this survey.

searching, and ❹ identifying criteria for including or excluding research. These aspects are defined below.

(1) The *objectives* of this survey are defined as follows:

O1 Provide the research community with a comprehensive catalog of DL kernel optimization methods, while tracing their development over time.

O2 Discuss directions for future research to extend the research on DL kernel optimization.

(2) The *research questions* in this survey are as follows:

RQ1 How are LLM-based kernel optimization methods currently evaluated? (response in Section III)

RQ2 How do existing approaches perform kernel optimization using LLMs? (response in Section IV)

RQ3 What roadmap can guide future research in LLM-based kernel optimization? (response in Section V)

(3) We considered several major publication platforms, including the ACM Digital Library, IEEE Xplore, Springer, ScienceDirect, arXiv, and Google Scholar. The search strategy employed primary keywords combining *GPU/DL kernel* (or *operator*) with *generation* (or *optimization*), supplemented by additional terms such as *performance* and *system*.

(4) The resulting publications were systematically screened to identify the most relevant works. Initially, a total of 18,199 publications were considered, which were subjected to a preliminary screening based on titles and research fields, followed by an abstract-level relevance assessment. Duplicated records were then removed, and the remaining works were further filtered according to publication period to retain only post-LLM studies. A full-text screening was subsequently conducted to determine the final set of core papers. To further mitigate the risk of missing relevant studies, a snowballing search was performed on the selected core papers to identify additional candidates. These newly collected works were screened using the same multi-stage procedure, including abstract-level filtering and full-text examination. Finally, supplementary open-source repositories and technical blogs were incorporated to complement the academic literature, resulting in a curated corpus of 36 works, as illustrated in Figure 1.

The rest of this survey is organized as follows. Section III reviews existing performance benchmarking methodologies. Section IV distinguishes between three major kernel optimization methodologies: single-agent systems, multi-agent systems, and training-based approaches, and organizes the surveyed methods accordingly. Section V outlines a forward-looking research roadmap, drawing on key insights from manual kernel optimization efforts in pre-LLM research and advances in the general code generation domain.

To support reproducibility and facilitate future research, we also provide a curated GitHub repository that catalogs all surveyed works and related resources. The repository is publicly accessible at: https://github.com/luckily268/Awesome-GPU-Kernel-Optimization.

II. Background and Preliminaries

Deep learning is intrinsically characterized by computationally intensive, massively parallel tensor operations, most notably large-scale matrix multiplications and convolutions. This computational paradigm is fundamentally misaligned with the architecture of general-purpose CPUs, which are designed primarily for sequential execution, complex control logic, and low-latency single-thread performance.

To address this architectural gap, specialized hardware accelerators such as GPUs, TPUs, and NPUs have been developed. Their designs prioritize massive parallelism through

Table I
CROSS-VENDOR MAPPING OF GPU ARCHITECTURAL TERMINOLOGY

Component	NVIDIA	AMD	Intel
Compute Unit	SM (Streaming Multiprocessor)	CU (Compute Unit)	Xe-core
SIMT/SIMD Execution Group	Warp (32 threads)	Wavefront (64 threads on GCN/CDNA; 32/64 on RDNA)	Thread Group (runtime-selected size; SIMD width 8/16/32)
Scratchpad Memory	Shared Memory	LDS (Local Data Share)	SLM (Shared Local Memory)
Matrix Accelerator	Tensor Core (FP8 / FP16 / BF16 / INT8)	MFMA (Matrix Core)	XMX (Xe Matrix Extensions)
Cache Strategy	Configurable L1/Shared Memory partition	Vector L0 + L1 + L2 hierarchy	Configurable L1/SLM Cache/SRAM partition

thousands of streamlined cores capable of executing identical operations across large datasets with high efficiency. In the following, we provide a brief overview of the hardware design of accelerators and their operating models.

A. Hardware Design of Accelerators

The design of modern hardware accelerators for DL is primarily organized around two fundamental key dimensions: the *Compute Cores* and the *Memory Hierarchy*. A schematic of this organization is shown in Figure 2.

The compute cores constitute the parallel arithmetic engines. While vendor-specific terminology varies (as detailed in Table I), such us NVIDIA's Streaming Multiprocessors (SMs), AMD's Compute Units (CUs), and Intel's Xe-cores, their underlying execution philosophy converges on a throughput-first paradigm. This is realized through wide SIMT (Single-Instruction, Multiple-Threads) or SIMD (Single-Instruction, Multiple-Data) pipelines, hardware-managed multithreading, and tightly coupled on-chip storage for thread state. For instance, an NVIDIA SM executes warps of 32 threads across its SIMT pipelines; AMD CUs primarily schedule wavefronts of 64 threads; and Intel Xe-cores employ explicit SIMD vector engines with thread-group scheduling. Each such unit typically integrates scalar ALUs (for thread-private operations), wide vector ALUs, dedicated load/store pipelines, and increasingly, specialized matrix-multiply accelerators—such as NVIDIA's Tensor Cores, AMD's Matrix Cores (executing MFMA instructions), and Intel's Xe Matrix Extensions (XMX). A hardware scheduler (warp/wavefront/thread-group scheduler) dynamically interleaves these thread groups to hide memory-access latency and maximize functional-unit utilization.

The memory hierarchy serves as the foundational data-supply architecture designed to meet the immense operand-throughput demands of parallel compute cores. It is structured as a multi-tiered system, spanning from high-capacity, high-bandwidth, yet high-latency off-chip memory (e.g., HBM or GDDR) down to low-latency, software-managed on-chip storage such as scratchpad memory (e.g., NVIDIA's shared memory, AMD's LDS, or Intel's SLM) and dedicated L1 caches. This layered design is essential to amortize the cost of data movement, facilitate efficient data reuse, and thus determine the practical fraction of theoretical peak compute throughput that can be achieved. Consequently, the orchestration of data movement across this hierarchy, whether via cache policies, explicit DMA operations, or compiler-directed scratchpad allocation, is as critical to overall accelerator performance as the design of the computational units themselves.

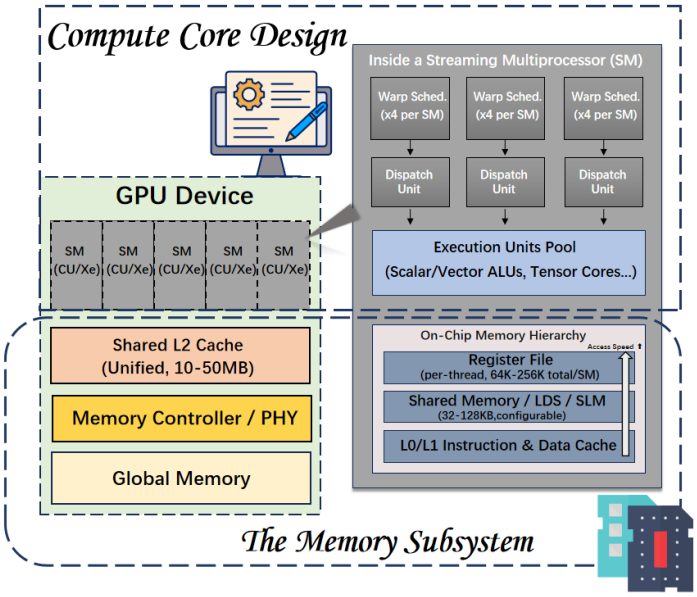

Figure 2. GPU architecture overview. The diagram illustrates the two primary subsystems of a modern GPU. Up: Compute Core Design — The GPU comprises an array of parallel Streaming Multiprocessors (SMs, also called Compute Units or Xe-cores) that access off-chip Global Memory via a shared, unified L2 Cache and Memory Controllers/PHY. Down: Memory Subsystem within an SM — Each SM contains multiple Warp Schedulers and Dispatch Units that issue instructions to an Execution Units Pool (with scalar/vector ALUs and Tensor Cores). Data is supplied through a dedicated on-chip hierarchy: the fastest Register File, software-managed Shared Memory/LDS/SLM, and hardware-managed L0/L1 Caches.

B. Operating Models of Accelerators

The performance of hardware accelerators is largely influenced by the compute kernels executed on their cores. These kernels are programmed directly using architecture-specific, low-level languages and instruction sets. For NVIDIA GPUs, this involves CUDA C++ and PTX assembly; for AMD GPUs, HIP C++ and the ROCm toolchain; and for specialized units like Tensor Cores, vendor-provided intrinsic functions or microcode-level APIs. Programming at this level grants experts precise control over thread scheduling, memory access

patterns, and execution pipelines, enabling the handcrafted optimization of kernels that can approach the theoretical peak throughput of the hardware.

However, developing kernels using such low-level programming remains challenging for most practitioners. To facilitate easier development, modern DL frameworks and software stacks provide developers with progressively higher-level programming interfaces for operating hardware, rather than relying on low-level programming. For better illustration, we will trace the execution path of a computational operator through the PyTorch stack on a CUDA-capable GPU. The following dissects this stack layer by layer, showing how a user's Python code is transformed, optimized, and ultimately executed as efficient, hardware-native kernels.

The typical execution path for an operator through this stack can be broken down into four stages: ❶ Python API Invocation, where the user's operator call is captured; ❷ C++ ATen Dispatching, which dynamically selects the appropriate backend kernel; ❸ CUDA Kernel Launch, where execution commands are prepared for the GPU; ❹ GPU Hardware Execution, where the kernel runs on parallel streaming multiprocessors. In addition, we illustrate the Triton path as an alternative execution path extending the traditional ATen flow for comparison.

1) Python API Invocation: First, developers write Python scripts that call PyTorch operators, for example:

```
import torch

# Initialize tensors
a = torch.tensor([1.0, 2.0], device='cuda')
b = torch.tensor([3.0, 4.0], device='cuda')

# Element-wise addition on GPU
c = a + b
```

Listing 1. GPU Tensor Operations in PyTorch

When the Python interpreter executes this code, it calls PyTorch's Python API. PyTorch's Python API is bound to the underlying C++ implementation through PyBind11.

When calling `a + b`, it actually invokes the `__add__` method of the `torch.Tensor` object, which calls the underlying C++ function through PyBind11.

2) C++ ATen Dispatching: With the C++ function call and tensor metadata from the previous stage, PyTorch's ATen tensor library takes over as the central dispatch engine. ATen's role is to dynamically route each tensor operation under PyTorch's dynamic execution model to the appropriate low-level kernel implementation, thereby decoupling the flexible, user-facing Python API from hardware-specific code. It provides a unified and extensible interface that bridges PyTorch's runtime-dynamic graph to a wide range of backend implementations, including CPU, CUDA, XPU, and ROCm.

When an operation such as `a + b` arrives from the Python layer, ATen inspects the tensor metadata, device, data type, shape, and other attributes, and dynamically dispatches the call to the corresponding backend kernel. For example, for CUDA-resident float tensors, ATen selects the CUDA-optimized add kernel to execute the actual computation.

3) Kernel Launch: With the specific kernel function pointer, device pointers to tensor data, and calculated launch parameters from the previous stage, ATen calculates the optimal GPU thread configuration, including grid dimensions, thread-block dimensions, and other parameters, then asynchronously launches the kernel through the CUDA runtime. Specifically, *add_kernel<<< grid, block, 0, stream>>>*, the CUDA runtime places the kernel launch command and parameters into the command queue of the specified CUDA stream, and the CPU immediately returns to continue execution without waiting for GPU computation to complete.

4) GPU Hardware Execution: Once the kernel launch command reaches the GPU, its execution follows a fixed hardware pipeline: Grid and Thread-Block Allocation distributes work to SMs; Warp Scheduling issues thread groups to hide instruction latency; Memory Access loads data from the register file, shared memory, or global memory; Computation Execution performs arithmetic in CUDA cores or specialized units; and finally, Result Write-back stores outcomes to designated memory locations. This structured flow enables the massive data parallelism that defines GPU acceleration.

While the multi-layered PyTorch stack provides accessible abstractions that spare most developers from writing CUDA C++, which is a complex and error-prone process, this convenience comes at the cost of reduced flexibility for fine-grained kernel optimization. To increase flexibility while maintaining low development overhead, Triton [16] was introduced as a high-level, Python-like language specifically designed for GPU kernel development. Triton provides a more intuitive and high-level programming interface, allowing developers to express complex GPU computation patterns without writing low-level CUDA code. At the same time, it offers fine-grained control over hardware resources, such as memory hierarchy, thread/block mapping, and vectorized operations, giving expert users the ability to tune kernels for maximum performance. This combination of ease of use and low-level control makes Triton a particularly promising tool for both rapid prototyping and performance-critical kernel development.

Specifically, practitioners can write Triton kernels as valid Python functions, which follow a Python-embedded domain-specific language (DSL) designed for GPU programming. Practitioners explicitly annotate such functions with `@triton.jit`, which registers them with the Triton runtime as parametric GPU kernel templates rather than executable Python code. Upon kernel invocation, the Python frontend extracts tensor metadata, including shapes, strides, data types, and device information, from the input tensors. This metadata is used by the Triton compiler to specialize the kernel for a specific execution configuration. During compilation, Triton lowers the annotated kernel through a multi-level intermediate representation (IR) pipeline and instantiates a target backend (e.g., CUDA or ROCm), applying a sequence of backend-specific optimization and code-generation passes. The resulting PTX code is then JIT-compiled by the GPU driver into native machine instructions (SASS), which are subsequently launched for execution on the GPU. The compilation and execution pipeline can be summarized as:

$$\text{Triton kernel} \xrightarrow{\text{Triton Compiler}} \text{PTX} \xrightarrow{\text{JIT}} \text{SASS} \xrightarrow{\text{Kernel Launch}} \text{GPU Execution}$$

C. Efficiency Bottlenecks of Kernels in DL Software Stacks

While frameworks such as PyTorch can automatically generate and dispatch computational kernels, these default implementations often fail to achieve peak hardware performance due to an inherent trade-off between generality and specialization. The underlying reasons are multifaceted.

❶ While PyTorch's automation successfully abstracts hardware complexity and ensures functional correctness, the kernels it employs, typically sourced from vendor libraries like cuBLAS or generated just-in-time, are designed as general-purpose implementations. These kernels must support a wide range of tensor shapes, data types, and hardware variants, which inherently prevents them from exploiting the full performance potential of any specific workload. For example, a general matrix multiplication kernel cannot assume fixed dimensions or memory layouts, and therefore cannot apply aggressive, shape-specific optimizations such as tailored loop tiling, unrolling, or memory-access patterns.

❷ Moreover, PyTorch's dynamic execution model defers key optimization decisions until runtime, limiting the scope for deep static analysis and compilation. This constraint stems from its core design: tensor properties such as shape, data type, and device are only known during execution, which prevents the compiler from making aggressive, irreversible optimizations upfront. For instance, while a matrix multiplication kernel generated at runtime must accommodate arbitrary tensor shapes, a pre-optimized kernel can be specialized for a fixed size, such as 256×256. This specialization enables compile-time optimizations unavailable in the dynamic path: explicit orchestration of data movement across the memory hierarchy (global → shared → registers), fine-tuning of thread-block and grid dimensions to maximize occupancy and latency hiding, and precise alignment of data layouts and instruction sequences to exploit specialized hardware such as Tensor Cores or Matrix Cores. Consequently, specialized kernels, whether manually engineered or auto-tuned, can achieve substantially higher hardware utilization for their target workloads than the general-purpose kernels dispatched through PyTorch's default stack, often translating to measurable, multi-fold speedups across training and inference pipelines.

The significant performance bottleneck of current general-purpose computational kernels, which fail to fully exploit the underlying hardware's computational potential, has become a critical limitation in deep learning systems. This persistent gap has motivated the development of numerous kernel generation and optimization techniques.

D. Problem Definition

DL kernel generation and optimization aim to automatically produce high-performance compute kernels for a target tensor computation and to further refine their execution behavior through schedule- and hardware-level optimizations. Given an operator-level specification S (e.g., a PyTorch operator, an intermediate representation, or a natural-language description), the system must (i) generate a kernel implementation K in a backend language $B \in \{\text{CUDA}, \text{Triton}, \text{HIP}, \text{MLIR-based backends}, \ldots\}$, and (ii) optimize its schedule, memory hierarchy usage, tiling strategy, parallelization mapping, and compilation parameters to achieve maximal runtime performance on a target hardware platform H.

This problem manifests through multiple input–output pathways, *including but not limited to*:

- natural language descriptions → CUDA kernels,
- PyTorch operator specifications (or other high-level tensor programs) → CUDA kernels,
- PyTorch operator specifications (or other high-level tensor programs) → Triton kernels.

Formally, DL kernel generation and optimization seek to solve:

$$K^* = \arg \max_{\substack{K \in \mathcal{G}(S,B), \\ \theta \in \mathcal{O}(K)}} P(K, \theta, H), \text{ such that } K \models S$$

where:

- $\mathcal{G}(S, B)$ is the space of candidate kernel implementations generated from specification S in backend B,
- $\mathcal{O}(K)$ is the optimization space over schedules, tilings, memory layouts, unrolling factors, launch configurations, and compiler parameters,
- $P(K, \theta, H)$ denotes the performance of kernel K under optimization parameters θ on hardware H.

E. Requirements for Kernel Optimization Methods

Here, we summarize a set of requirements that kernel optimization methods are expected to satisfy.

• **Functional Correctness.** The kernel produced by the kernel optimization method is expected to be functionally correct.

• **Acceleration.** The kernel generated by the method should make full use of hardware accelerators to maximize computational acceleration.

• **Robustness.** The kernel generated by the method should demonstrate stable correctness and performance under edge-case or challenging settings, including extreme tensor shapes, atypical data types, and adversarial or rare input distributions.

• **Efficiency.** Kernel optimization methods should minimize kernel optimization time overhead.

• **Versatility.** Kernel optimization methods must handle a broad range of operators, such as convolution and attention, along with diverse tensor shapes and numeric data types, thereby ensuring generalization beyond narrow, handcrafted examples [17–19].

• **Cross-Architecture Portability.** Kernel optimization methods should support multiple hardware architectures and software backends with minimal modification.

• **Reproducibility.** Kernel optimization methods should produce kernels whose behavior, both correctness and performance, is reproducible under well-controlled settings.

III. EVALUATION

This section addresses **RQ1** by systematically reviewing how to evaluate LLM-based kernel generation methods. In this section, we first summarize existing benchmarks and metrics commonly used in the literature, as in Table II and Table III. We then outline the challenges and present a roadmap.

A. Existing Evaluation Benchmarks

KernelBench [17] and **TritonBench** [20] initiate the design of systematic evaluation benchmarks and metrics for assessing both the functional correctness and acceleration performance of LLM-generated kernels.

KernelBench establishes a foundational evaluation benchmark for CUDA kernel optimization in DL workloads. It consists of 270 programming tasks spanning multiple difficulty levels (L1–L4) and adopts the $fast_p$ metric, which measures the proportion of generated kernels that are both functionally correct and achieve at least $p\times$ speedup over PyTorch baselines.

TritonBench focuses on evaluating kernel generation using the Triton DSL and provides a complementary benchmark suite of 350 real-world operators curated from GitHub and PyTorch repositories. It evaluates functional correctness using Pass@K and measures runtime performance using Speedup relative to optimized baselines.

However, these initial benchmarks expose several critical limitations. ❶ They remain confined to NVIDIA's ecosystem (CUDA or Triton), leaving other hardware platforms unexamined. ❷ They evaluate kernels only on a limited set of input shapes and configurations, offering little insight into robustness across diverse workloads and runtime variations.

To overcome these shortcomings, the community soon proposed two new evaluation benchmarks. **MultiKernelBench** [18] presented by wen et al. directly targeted the first limitation of platform specificity. It introduced the first benchmark supporting kernel generation for multiple backends: CUDA, AscendC (Huawei NPU), and Pallas (Google TPU). Its core innovation was a modular backend abstraction layer that decoupled evaluation logic from platform-specific toolchains, enabling fair comparison across diverse hardware.

Meanwhile, **NPUEval** [3] establishes a new benchmark for evaluating LLMs' ability to generate vectorized kernel code for NPUs. More than just a dataset, it provides a complete open-source evaluation harness with cycle-accurate performance metrics.

Concurrently, the **robust-kbench** [19] addresses the second limitation. It evaluates kernel correctness across diverse settings, supports both forward and backward kernel optimization, and is designed for realistic downstream applications.

Additionally, the **BackendBench** framework [21] proposed by Meta further systematically validate the functional correctness and performance of kernels generated by LLMs in real deployment. The framework conducts comprehensive functional correctness tests on 271 operators based on TorchBench and PyTorch's OpInfo, ensuring consistency with standard implementations. Meanwhile, using real tensor shapes from models in Huggingface, it performs performance testing on 124 commonly used operators to evaluate their execution efficiency under practical workloads. Furthermore, BackendBench introduces a *success rate across attempts* metric to assess the stability of the generation process.

B. Challenges in Benchmarking

Kernel generation benchmarking faces multiple challenges at both the individual kernel level and the benchmark suite level. Following the requirements outlined in subsection II-E, we organize these challenges accordingly.

Functional Correctness. Many existing kernel benchmarks provide reference implementations or fixed input sets to evaluate correctness. While these allow basic verification, they typically cover only a limited range of shapes, data types, and input distributions. This limited coverage means that kernels may appear correct under benchmark conditions but could fail in more diverse or realistic scenarios. Addressing this limitation is challenging because designing inputs that comprehensively reflect real workloads is nontrivial.

Acceleration Performance. Existing benchmarks often measure runtime to evaluate kernel acceleration, but achieving consistent and reliable measurements is difficult due to hardware variability, warm-up effects, and differences across frameworks and backends. These factors can make it hard to determine whether observed performance improvements reflect true optimization or are influenced by external conditions. This highlights the need for standardized execution environments and well-defined measurement protocols, including warm-up runs and iteration counts, to ensure reproducible results.

Robustness. While many benchmarks focus on typical inputs, kernels must also maintain correctness and stable performance under extreme or rare edge-case scenarios, such as unusual tensor shapes, atypical data distributions, or boundary batch sizes. Existing evaluations often overlook these conditions, leaving kernels vulnerable to failures or performance degradation in challenging situations.

Efficiency. Existing evaluations rarely quantify the efficiency of kernel generation and optimization pipelines, leaving methods that are slow or resource-intensive insufficiently assessed.

Versatility. Existing benchmarks, such as NPUeval and ComputeEval, often focus on a limited set of operators, shapes, and data types. While these benchmarks provide useful snapshots of kernel behavior, their narrow scope makes it difficult to draw general conclusions about performance, correctness, or robustness across a full spectrum of workloads. This limited coverage presents a key challenge because kernels may perform well on benchmarked operators but fail or underperform on untested ones.

Cross-Architecture Portability. Many existing benchmarks are tailored to a single hardware platform, which limits the ability to compare kernel performance or correctness across different architectures. This focus on a single platform makes it challenging to assess whether optimizations generalize beyond the target hardware, and it reduces the relevance of benchmark results for broader deployment.

Reproducibility. Many existing benchmarks report results that can vary significantly depending on execution environments, framework versions, or hardware configurations. This variability makes it difficult to reliably compare kernel performance or correctness across experiments and over time, and it can obscure the true impact of optimization techniques. Addressing this issue is challenging because even minor differences in system setup or runtime conditions can affect outcomes.

C. Roadmap for Advancing Benchmarking

This roadmap outlines strategic directions to improve benchmark design.

For functional correctness, benchmarks should move beyond fixed or narrowly scoped input configurations and adopt multi-dimensional validation across diverse tensor shapes, data types, and initialization schemes. For acceleration performance, benchmarks should emphasize standardized and reproducible measurement protocols. This includes clearly defined warm-up procedures, iteration counts, and isolation of execution environments, so that reported speedups more accurately reflect genuine optimization effects rather than artifacts of runtime variability or framework-specific behavior. For robustness, optimized kernels should be evaluated under extreme or uncommon conditions, such as irregular tensor shapes, in order to expose brittle optimization strategies that perform well only under nominal settings.

For efficiency, evaluation should extend beyond the quality of the generated kernels themselves to include the cost of the generation and optimization process. Benchmarks should measure factors such as generation latency, compilation overhead, search or iteration budgets, and overall resource consumption, providing a more realistic assessment of practical usability. For versatility, benchmarks should expand the diversity of evaluated kernels by including operators drawn from a wide range of model families and application domains. Leveraging real workloads and tensor shapes from modern deep learning models can help ensure that benchmark results remain representative as workloads continue to evolve. For cross-architecture portability, benchmarks should adopt modular and backend-agnostic designs that enable evaluation across heterogeneous hardware platforms, including GPUs, NPUs, and emerging accelerators. Such designs can expose architectural biases in kernel generation models and encourage the development of more portable optimization strategies. For reproducibility, benchmarks should provide open, well-documented evaluation harnesses and reference environments, enabling results to be reliably reproduced and compared across different systems, software stacks, and time periods.

The emergence of more comprehensive benchmarks could serve both as a testbed for existing methods and as a feedback mechanism to improve future kernel generation models.

IV. Kernel Generation and Optimization Techniques

This section addresses **RQ2** by reviewing how existing approaches perform kernel generation and optimization using LLMs. Existing kernel optimization approaches can be broadly classified into three categories, as shown in Figure 3: single-agent, multi-agent, and training-based methods. This section presents a detailed overview of each category. These works are summarized in Table IV.

Figure 3. Distribution of LLM-driven GPU kernel generation and optimization research. Percentages are calculated over the 37 surveyed post-LLM kernel generation and optimization papers. Works focusing primarily on mobile systems [4] are excluded from the statistics. KernelBench [17] and AI CUDA Engineer / robust-kBench [19] span multiple methodological categories and are therefore counted separately. In addition, partial code releases and documentation-only resources included in Figure 1 are not considered in this statistical count.

A. Single-Agent Systems

The initial wave of research on LLM-driven kernel optimization primarily centered around single-agent systems. The evolution within this category demonstrates a clear trajectory: starting from evaluating basic prompting efficacy, integrating with verifier and profiler, and culminating in reformulating kernel optimization as a structured optimization problem.

In the early exploration of automated kernel generation with LLMs, Ouyang et al. [17] introduce KernelBench and conduct a series of pilot experiments. They first adopt a one-shot prompting approach and evaluate several state-of-the-art models, including GPT-4o [23], DeepSeek-R1 [24], and Llama [25]. The results reveal that even the best models can only outperform the PyTorch baseline in fewer than 20% of the tasks under one-shot generation. While reasoning-enhanced models exhibit fewer execution errors, they still struggle with functional correctness. When tested across multiple NVIDIA GPU platforms (L40S, A100, H100, T4, etc.), the performance of generated kernels varies considerably, indicating limited model adaptability to hardware-specific characteristics. Subsequently, they also experiment with feedback-driven optimization and knowledge-augmented prompting. Their findings demonstrate that iterative refinement incorporating execution and manual feedback effectively helps models correct errors and discover more efficient implementations. When provided with relevant

Table II
OVERVIEW OF BENCHMARKS FOR LLM-BASED GPU KERNEL GENERATION

Benchmark	Institution	Date	Core Task	Dataset Composition	Metrics
KernelBench [17]	Stanford	2025.02	Torch → CUDA	270 tasks: L1 (100), L2 (100), L3 (50), L4 (20)	fast_p
TritonBench [20]	Tianjin Univ.	2025.02	Torch → Triton	350 tasks: GitHub (184), PyTorch (166)	Pass@K, Speedup
Compute-eval [22]	NVIDIA	2025.04	NL → CUDA	128 programming problems	Pass@K
NPUEval [3]	AMD	2025.07	NL/Spec → Vectorized C++ for NPU	102 common ML operators for AMD NPU	Functional Correctness, Cycle-accurate, Vectorization Score
MultiKernel-Bench [18]	Nanjing Univ.	2025.07	Torch → Multi-backend	285 operators across CUDA, AscendC, Pallas	Pass@K, Compilation@K, SpeedUp_α@K
BackendBench [21]	Meta	2025.07	Torch → Triton	271 ops (correctness), 124 ops (performance)	Correctness, Speedup
robust-kbench [19]	Sakana AI	2025.09	Torch → CUDA	Tasks with multi-inits, shapes, forward/backward passes	Correctness, Speedup, Generalization

Table III
EVALUATION DIMENSIONS COVERAGE ACROSS GPU KERNEL GENERATION BENCHMARKS

Benchmark	Functional Correctness	Acceleration Performance	Robustness	Efficiency	Versatility	Cross-Architecture Portability	Reproducibility
KernelBench	✓	✓			✓		✓
TritonBench	✓	✓		✓			
NPUEval	✓	✓	✓	✓		✓	
MultiKernelBench	✓	✓			✓	✓	
robust-kbench	✓	✓	✓		✓		

hints, models attempt to employ more advanced optimization strategies, such as shared memory or tensor core instructions, though this often increases the risk of compilation and runtime failures.

Following the agent design in KernelBench, Chen et al. [26] has developed a new workflow that combines the DeepSeek-R1 model with verifier in a closed-loop fashion to generate optimized attention kernels. The workflow begins with a manual prompt, and the DeepSeek-R1 model generates the initial GPU kernel. The verifier, running on an NVIDIA H100 GPU, analyzes the generated kernel and creates new prompts that are fed back to the model. This closed-loop approach iteratively refines the code generation process and achieves 100% numerical correctness on Level-1 problems and 96% on Level-2 problems. These results demonstrate the potential of using advanced models like DeepSeek-R1 with increased computational resources during inference to generate high-performance GPU kernels.

To further explore how to enable LLMs to generate high-quality kernel code, Brabec et al. [27] from Charles University and other institutions systematically evaluate the capability of reasoning LLMs to produce optimized CUDA code through three well-known CUDA assignments. By introducing a *tutoring* mechanism (providing more detailed optimization hints and algorithmic descriptions in the prompts), they find that the quality of generated code can be significantly improved. For simpler CUDA tasks like computing histogram, where the optimization space is relatively straightforward, appropriate suggestions alone enable the model to autonomously complete the optimization. However, for more complex problems like k-nearest neighbors, which require intricate parallel algorithm design, the models often fail to produce correct solutions without explicit, step-by-step guidance. The study reveals that while LLMs excel at following clear instructions, they struggle to make high-level optimization decisions independently when lacking adequate guidance. Furthermore, the models exhibit limitations in selecting algorithmic hyperparameters, underscoring the continued importance of integrating performance evaluation, or even auto-tuning with LLM-based code generation.

CUDA-LLM presented by Chen et al. [28] integrates a **FSR** (Feature Search and Reinforcement) framework that places the LLM in a foundational workflow ("natural language → candidate generation → validation → performance optimization → prompt update"). Concretely, CUDA-LLM decomposes the verifier used in prior work such as Chen et al. [26] into three separated components: a *Compilation Verifier* to ensure syntactic and build correctness, a *Function Validator* to check the functional correctness of the kernel, and a *Performance Profiler* to evaluate on-GPU execution efficiency. This structured verifier design enables CUDA-LLM to form a feedback signal over compilation validity, functional correctness, and runtime performance, thereby supporting the iterative reinforcement optimization loop of FSR. However, the model itself is not trained to internalize generalizable tool-usage behaviors, and directly motivates subsequent training-based approaches in subsection IV-C.

Guo et al. [29] propose **EvoENGINEER**, a framework

that abstracts LLM-based kernel optimization into a structured evolutionary code search process. Rather than introducing another ad hoc workflow, EvoENGINEER organizes code evolution into two orthogonal components: traverse techniques for navigation strategies in the discrete code space and population management for maintaining and selecting candidate solutions. This abstraction facilitates more effective independent analysis and systematic comparison of different evolution strategies. Based on this framework, they instantiate three representative variants: *EvoENGINEER-Free* that utilizes only task context, *EvoEngineer-Insight* that leverages optimization insights and *EvoENGINEER-Full* that integrates both historical solutions, forming a spectrum of progressively richer information integration and population preservation strategies. Evaluated on 91 real-world CUDA kernels, EvoEngineer achieves a principled balance between performance and correctness, with the highest averaged median speedup of 2.72× over baseline CUDA kernels and a code validityrate of 69.8%, establishing a principled and reusable foundation for evolutionary kernel optimization.

From a decision theoretic perspective, Ran et al. [30] introduce **KERNELBAND**, which reformulates kernel optimization as a hierarchical sequential decision problem under performance uncertainty. Instead of treating kernel refinement as ad-hoc iteration, KERNELBAND models kernel candidate selection and optimization strategy application as two coordinated bandit layers, guided by profiling signals. The framework incorporates runtime behavior clustering to reduce redundant exploration across similar kernels and leverages hardware profiling feedback to bias the search toward promising optimization directions. Evaluated on TritonBench [20], KERNELBAND consistently outperforms state-of-the-art baselines, achieving higher kernel efficiency with substantially fewer tokens and exhibiting strong scalability without saturation as more computational resources are available.

B. Multi-Agent Systems

Beyond single-agent paradigms, more works advance LLM-based kernel optimization by adopting multi-agent architectures, where coordinated interactions among specialized LLM agents govern the optimization process in place of predefined pipelines.

The **GPU Kernel Scientist** framework proposed by Andrews and Witteveen [31] represents an early instantiation of multi-agent systems for GPU kernel optimization. It casts optimization as a scientific discovery process following a *hypothesis–experiment–validation* loop, executed by a fixed set of roles (Designer, Writer, Tester) under evolutionary selection. This formulation enables exploration of unfamiliar or poorly documented hardware (e.g., AMD MI300) with minimal prior expertise. However, the system relies on serial execution-time evaluation as its sole feedback signal, lacks profiler-level guidance, and scales optimization primarily through repeated iterations, leading to slow convergence.

Building on this paradigm, Wang et al. [32] propose **GEAK**, which introduces a redesigned agentic optimization system for CUDA kernel generation. GEAK organizes kernel optimization into four coordinated agent roles: *Generator*, *Evaluator*, *Reflector*, and *Optimizer*, forming a closed feedback pipeline: the Evaluator performs cascaded correctness and performance checks, the Reflector analyzes error traces and failures, and the Optimizer formulates targeted refinement strategies that are fed back to the Generator for subsequent iterations. This fine-grained decomposition enables scalable parallel exploration via inference-time compute scaling, rather than relying solely on serial evolutionary iteration. Moreover, GEAK incorporates Reflexion-style feedback loops, allowing failed or suboptimal kernels to be analyzed and revised through error tracing and reflective reasoning. These design choices makes GEAK better suited for large kernel spaces and performance-sensitive workloads. In addition, GEAK introduces AMD-focused benchmark suites (ROCm Triton Benchmark), enabling rigorous cross-platform evaluation that was absent in earlier systems.

Mishra and Nangia [33] take a fundamentally different, search-oriented view of multi-agent collaboration in *"How Many Agents to Beat PyTorch?"*. They introduce a central *Orchestrator* that manages a branching search process over parallel kernel hypotheses, casting kernel optimization as a structured tree search in the discrete code space. Within this orchestrated framework, a *Reasoner-Agent* proposes multiple optimization strategies in natural language, which are instantiated in parallel by a *Synthesis-Agent* into distinct kernel variants. Dedicated *Compile-Agent* and *Correctness-Agent* aggressively prune invalid or incorrect candidates before on-GPU performance evaluation, where surviving kernels compete and the winners seed the next search round. By controlling branching, pruning, and termination, the *Orchestrator* prevents premature convergence and infinite local refinement loops, enabling large-scale parallel exploration under inference-time compute scaling. Evaluated on NVIDIA H100 GPUs, the framework achieves substantial speedups (e.g., 4.0× for softmax), demonstrating that orchestrated multi-agent search can surpass both monolithic agents and PyTorch baselines when sufficient compute budget is available.

Additionally, unlike prior systems that primarily generate optimized kernels from scratch, Wei et al. [34] propose **Astra**, shifting the problem setting toward optimizing existing CUDA kernels from SGLang [35], which is a widely deployed LLM serving framework. Astra organizes the optimization loop into four specialized agents. A *Testing Agent* constructs correctness test suites and validates candidate kernels, while a *Profiling Agent* measures execution time and memory behavior to provide hardware-level performance feedback. A *Planning Agent* jointly reasons over correctness and profiling signals to propose targeted transformations, and a *Coding Agent* applies these plans to synthesize new kernel implementations. To enable direct optimization of SGLang's highly interdependent kernels, Astra further introduces a pre-/post-processing pipeline that extracts kernels into stand-alone forms for optimization and subsequently reintegrates optimized implementations back into the full framework for validation and benchmarking. This design allows Astra to report speedups relative to the original production kernels while preserving compatibility with the original framework. Evaluated on real SGLang kernels, Astra achieves consistent speedups over single-agent baselines under zero-shot prompting, highlighting

the practical potential of multi-agent systems for maintaining and optimizing production GPU code.

Beyond coordinating specialized agents for code generation, testing, and profiling, the **ai cuda engineer** framework proposed by Lange et al. [19] introduces a dedicated LLM-based verifier as a central design innovation. Its key improvement lies in treating correctness verification as a loop-internal, learnable optimization signal rather than a purely post-hoc execution filter. By performing early "soft verification" prior to hardware execution, the verifier prunes obviously incorrect candidates at the input stage rather than relying on expensive post-execution result checking, thereby enabling deeper and more aggressive exploration of the kernel search space. Moreover, ai cuda engineer integrates error summarization and in-context improvement into this verification loop, forming a closed-loop evolutionary workflow for translating PyTorch operators into optimized CUDA kernels and supporting complex transformations such as multi-operator fusion.

Furthermore, **STARK** proposed by Dong et al. [36] advances prior LLM-based kernel optimizers by redesigning kernel refinement as a tightly coordinated multi-agent process with strategic tree search over persistent memory. STARK decomposes optimization into specialized planning, coding, and debugging agents, and introduces grounded instructions and dynamic context windows to translate high-level strategies into precise, localized CUDA code edits. Grounded instructions anchor planned transformations to concrete code spans, specifying where and how to apply each optimization, while dynamic context windows expose different historical attempts and feedback to specific agents, enabling experience-guided planning, implementation, and debugging. This design tightly couples strategic reasoning with low-level execution and balances exploration and exploitation to systematically navigate the code space, mitigating common failure modes such as incoherent refinements and myopic local search. Evaluated on KernelBench, STARK achieves substantially higher success rates and runtime speedups (up to $10\times$–$16\times$), particularly on kernels where baseline agents struggle to produce valid implementations.

Meanwhile, Wang and the PyTorch Team at Meta [37] produce **KernelFalcon**, which organizes kernel synthesis into a deterministic, orchestrated agent pipeline with decomposition, parallel exploration, and execution-based verification. Its workflow is decomposed into specialized agents responsible for operator fusion, subgraph extraction, Triton kernel synthesis, and end-to-end numerical validation, coordinated by a central Orchestrator that manages delegation, failure handling, and early-stop parallel search. Crucially, KernelFalcon adopts a verifier-first loop: candidate kernels are compiled and executed against PyTorch references, and the system early-exits upon discovering numerically correct implementations, enabling parallel exploration of diverse kernel realizations while preserving full PyTorch semantics. KernelFalcon is the first known open agentic system to achieve 100% correctness across all 250 L1/L2/L3 KernelBench tasks, demonstrating the effectiveness of deeply orchestrated, verification-driven agent pipelines for reliable kernel synthesis.

In contrast to large, highly structured multi-agent frameworks, Zhang et al. [38] propose **CudaForge**, a lightweight dual-agent system that separates kernel generation and evaluation into a Coder–Judge loop. The Coder generates CUDA kernel candidates based on task instructions and feedback from the Judge, while the Judge evaluates each candidate using correctness checks, runtime profiling, and hardware metrics (e.g., GPU specifications and Nsight Compute outputs) to identify bottlenecks and provide targeted optimization guidance. This iterative process allows the Coder to progressively refine kernels across multiple rounds, correcting errors and improving performance in a directed manner. By decoupling generation and evaluation, CudaForge achieves highest correctness rate and significant performance gains over baseline approaches on Kernelbench [17] while maintaining strong practical performance. These results highlight that even a minimalist agentic decomposition, when combined with iterative, hardware-aware feedback, can deliver meaningful gains in real-world kernel optimization.

Building on lightweight, profiling agentic refinement such as CudaForge, Lei et al. [39] further propose **PRAGMA**, a multi-agent framework that tightly integrates fine-grained hardware profiling into the LLM optimization loop. Not only does PRAGMA rely on correctness or coarse runtime feedback, but grounds iterative kernel refinement in detailed, hardware-aware performance signals collected from both GPU and CPU backends. PRAGMA employs a Profiler Agent to gather low-level metrics from diverse profiling tools, including Nsight Compute and Linux `perf`. A dedicated Conductor Agent then interprets these metrics, performs bottleneck classification, and distills them into high-level optimization hints. Guided by this feedback, the Coder Agent iteratively refines kernel implementations, while the system explicitly preserves historically best-performing variants and their profiling traces, enabling context-aware reasoning over evolving performance bottlenecks. Experimental results on KernelBench [17] demonstrate that PRAGMA consistently outperforms prior LLM-based approaches, achieving averaged speedups of 2.81× on CPU and 2.30×–4.50× on GPU, and up to 10.95× over baseline LLM-generated kernels. These results highlight the effectiveness of reasoning based on detailed profiling feedback and explicit bottleneck interpretation.

Li et al. [40] propose **TritonForge**, a framework that centers on a LLM optimization pipeline for Triton kernels. TritonForge incorporates specialized agents for test generation, kernel optimization, and fault-aware remediation, forming a multi-stage workflow that supports automated benchmarking, error correction, and iterative refinement. Profiling and code generation are performed in a closed loop until performance converges or a predefined iteration budget is reached, enabling TritonForge to progressively steer Triton kernels toward high-performance implementations without manual profiling expertise. Moreover, TritonForge also integrates NVIDIA Nsight Compute into the optimization loop to collect low-level hardware metrics, such as memory throughput, warp occupancy, and instruction stalls, and translates these profiling signals into structured feedback for the LLM. Based on this feedback, the model generates targeted code modifications, including changes to tiling strategies, memory layouts, and the insertion

of auto-tuning directives. While this profiling-guided loop enables TritonForge to progressively steer kernels toward higher performance without manual profiling expertise, its iterative search exhibits limited exploration efficiency: the LLM often revisits semantically similar but performance-neutral variants and tends to converge prematurely to shallow performance plateaus, reflecting the lack of gradient-like guidance in profiling feedback and motivating the need for stronger diversity control, adaptive stopping, and memory-augmented search in future designs.

Furthermore, Sereda et al. [41] introduce **KForge**, which is a platform-agnostic agentic framework. KForge is designed to operate across diverse accelerator backends. It combines a generation agent with a performance analysis agent that interprets profiling data from heterogeneous sources, including programmatic APIs and GUI-based tools. This work explores whether LLMs can generate kernel programs for multiple hardware accelerators, leveraging both algorithmic and hardware-specific optimizations. This separation between code synthesis and performance interpretation enables cross-platform knowledge transfer with minimal supervision. By requiring only a single example to target new hardware, KForge demonstrates that agentic optimization can generalize across fundamentally different parallel programming models, such as NVIDIA CUDA and Apple Metal.

Following this, Nagaitsev et al. [42] propose **PIKE**, a population-based multi-agent framework for iterative LLM-driven kernel optimization. PIKE models optimization as a population search process, where each agent corresponds to an independent LLM query and agents can be executed sequentially or in parallel using the same underlying model, forming a shared verification-driven evolutionary loop. The framework maintains a solution library storing the initial PyTorch model and validated candidates. At each iteration, existing solutions are selected as seeds, from which new kernels are generated via mutation or crossover. Candidate solutions are then compiled, functionally validated, and benchmarked, optionally refined by a dedicated Error Fixing Agent (EFA), and finally inserted back into the library. This loop repeats until convergence or a predefined budget is reached, and can be parallelized through island-based population structures. Within this framework, PIKE instantiates two representative strategies. PIKE-B (Branching Search) is an exploit-heavy, mutation-only strategy that duplicates the top-k elite solutions to form each new population, rapidly refining high-potential kernels under a single-island and short-term memory setting. In contrast, PIKE-O (OpenEvolve-based) emphasizes exploration through crossover across multiple elite solutions and island-based parallelism. Empirical results on the METR-refined variant of KernelBench [17] show that exploit-heavy strategies combined with EFA achieve more effective optimization trajectories, and that optimization step granularity is a key determinant of final performance.

Finally, industrial systems such as Huawei's **AKG** [43] framework illustrate how multi-agent principles can be scaled and integrated into production compiler stacks. The AIKG subproject adopts a role-specialized agent architecture, including Designer, Coder, Conductor, and Verifier agents, integrated with MLIR-based compilation and retrieval-augmented generation. Unlike research prototypes, AKG emphasizes extensibility, backend diversity, and workflow robustness, supporting multiple hardware targets such as Ascend accelerators. Notably, although Astra, KForge all exhibit forms of generalization, they generalize along fundamentally different dimensions. Astra [34] attempts to extend its ability to autonomously apply a diverse set of optimization patterns across different kernels, KForge [41] targets cross-platform hardware abstraction, while AKG achieves ecosystem-level generalization through deep integration with compiler infrastructures. These differences reflect distinct trade-offs between flexibility, control, and engineering complexity, and suggest that no single notion of generalization dominates across all optimization scenarios.

C. Training-based Methods

Beyond agent collaboration, a series of works adopt supervised fine-tuning (SFT) on curated datasets of optimized kernels, or reinforcement learning (RL) with execution-grounded rewards, enabling models to learn common optimization patterns.

Firstly, He and Yoneki [11] propose **CuAsmRL**, which represents a form of training-based optimization by directly operating on NVIDIA GPU SASS-level instruction schedules rather than high-level kernel code. CuAsmRL formulates SASS scheduling as an *assembly game*, where a reinforcement learning agent iteratively mutates instruction schedules starting from `-O3`-optimized baselines and receives throughput-oriented rewards obtained through empirical GPU execution. By learning to mimic expert-level manual scheduling behaviors, the model is able to automatically discover superior low-level schedules. However, this extreme specialization incurs substantial training cost, as reward signals must be obtained through repeated physical execution on GPUs. Moreover, the lack of accurate analytical performance models for SASS-level instructions, limits scalability and cross-domain generalization. Consequently, applying CuAsmRL to kernels from new domains still requires domain-specific retraining and manual verification.

KernelLLM [44] curated the KernelBook dataset and employs SFT for end-to-end Triton kernel generation. KernelLLM fine-tunes Llama-3.1-8B-Instruct on approximately 25,000 paired examples of PyTorch modules and their corresponding Triton kernel implementations, augmented with synthetically compiled samples generated via `torch.compile()` and curated code from TheStack [45]. The resulting dataset, KernelBook, provides structured supervision that explicitly aligns high-level PyTorch semantics with low-level Triton implementations. Trained using standard instruction-based SFT, KernelLLM translates PyTorch programs into Triton kernel candidates, which are validated through unit tests and pass@k sampling on KernelBench-Triton. Despite its relatively modest parameter scale, KernelLLM achieves competitive performance with significantly larger frontier models, highlighting the effectiveness of curated supervision in imparting GPU programming patterns.

However, as an imitation-based SFT approach, KernelLLM primarily inherits the optimization strategies present in the training corpus, limiting its ability to extrapolate beyond observed optimization patterns.

Kevin (Kernel Devin) [46] pioneers multi-turn reinforcement learning for CUDA kernel generation. For each task, Kevin samples multiple parallel trajectories, where kernels are iteratively refined over several turns. Each refinement turn consists of a chain-of-thought (CoT) reasoning step and a kernel generation step, where the CoT verbalizes intermediate optimization decisions, while the kernel generation step concretely implements these decisions into an updated CUDA kernel, and is treated as an individual training sample with execution-grounded rewards. To prevent context explosion, long CoTs are discarded while compact summaries of optimization actions, together with previously generated kernels and evaluation feedback, are retained to condition subsequent refinement turns. Evaluated on KernelBench [17], Kevin improves kernel correctness from 56% to 82% and increases mean speedup from $0.53\times$ to $1.10\times$ over the PyTorch Eager baseline. These results demonstrate that reinforcement learning can effectively train models to reason and optimize over a sequence of structured refinement steps. However, its evaluation is primarily conducted on NVIDIA A100 GPUs, leaving generalization to diverse hardware architectures as an open question.

Following Kevin, **CUDA-L1** introduced by Li et al. [47] includes three stages: Supervised Fine-Tuning with Data Augmentation, Self-Supervised Learning, and Contrastive Reinforcement Learning. The approach augments the training dataset with CUDA code variants generated by LLMs and fine-tunes the base model on executable and correct implementations to establish foundational CUDA knowledge. The model then iteratively generates CUDA kernels, validates their correctness and executability, and trains on successfully validated examples, enabling autonomous improvement without human supervision. Additionally, contrastive learning is employed with execution-time rewards, training the model to distinguish between faster and slower CUDA implementations, ultimately optimizing for superior performance. However, the CUDA-L1 approach relies on iterative generation, validation, and training cycles, which makes the whole process relatively time-consuming.

In the triton domain, Li et al. [48] introduce **AutoTriton** that represents the first dedicated RL-trained model for Triton kernel synthesis, combining SFT with Group Relative Policy Optimization (GRPO) reinforcement learning [49] under hybrid rewards based on rule and execution. Built on an 8B parameter architecture, AutoTriton first undergoes supervised fine-tuning on curated Triton examples, then is further optimized using the GRPO algorithm with a hybrid reward function that combines rule-based and execution-based feedback. AutoTriton demonstrates performance comparable to significantly larger frontier models (e.g., Claude-3.5 Sonnet and DeepSeek-R1) across five evaluation channels of TritonBench [20] and KernelBench [17]. The work highlights the effectiveness of RL in learning high-level Triton programming patterns and hardware-specific optimizations.

SwizzlePerf proposed by Tschand et al. [12] demonstrates that hardware topology–aware execution mapping policies can also be internalized into model parameters through training. Instead of generating full kernels, SwizzlePerf trains models to learn data–work–hardware swizzling policies by modeling GPU memory hierarchy and architectural topology (e.g., AMD XCD). The learned policies plan execution and storage mappings that optimize locality and cache utilization, effectively transferring human hardware–software co-design knowledge into learned optimization behaviors. Evaluations on ML and scientific kernels report speedups of up to $2.1\times$ and up to 70% improvements in L2 cache hit rate, illustrating that end-to-end training can internalize not only code-level but also hardware-mapping–level optimization strategies. However, its current scope mainly focuses on cache hierarchy optimization, leaving other hardware resources under-explored.

ConCuR (Concise CUDA Reasoning) proposed by Kong et al. [50] addresses the data bottleneck in LLM-driven kernel generation by introducing a data synthesis and curation pipeline. In the synthesis stage, 18,162 PyTorch programs from KernelBook are expanded via parallel reasoning-aware generation into 90,810 PyTorch–CoT–CUDA triplets, forming a large but noisy candidate pool. In the curation stage, ConCuR jointly selects samples based on reasoning conciseness, runtime speedup, and task-type balance, distilling 4,892 high-quality PyTorch–reasoning–CUDA triplets. Fine-tuning QwQ-32B on ConCuR yields **KernelCoder**, improving pass@1 correctness from 18% to 58% on Level-1 and from 17% to 59% on Level-2, while also significantly boosting fast1 performance.

The framework proposed by Nichols et al. [51] trains LLM to interact with performance analysis tools as part of the kernel optimization process. This approach fine-tunes models to perform tool-assisted reasoning at inference time, enabling them to iteratively formulate optimization hypotheses, invoke benchmarking and profiling tools, and refine kernel implementations through extended reasoning chains. The training procedure employs reinforcement learning objectives based on verifiable performance rewards, encouraging effective tool usage and measurable optimization improvements while avoiding the need for large-scale online benchmarking during training. By distilling optimization reasoning into compact models, the method amortizes performance engineering expertise into model parameters and enables efficient deployment. Empirical evaluations on GPU kernel benchmarks and real HPC applications demonstrate strong optimization capability, including a reported 17% kernel-level speedup that translates into a 3% end-to-end application improvement.

TritonRL proposed by Woo et al. [52] introduces an 8B-scale Triton-specialized language model trained with a hierarchical and verifiable reinforcement learning pipeline designed to achieve both high correctness and runtime performance while mitigating reward hacking. TritonRL combines supervised fine-tuning with DeepSeek-R1 distillation and a subsequent RL stage featuring fine-grained reward decomposition across correctness, efficiency, and style. Its verification framework integrates enhanced rule checks with LLM judges to construct robust, verifiable rewards, enabling reliable diagnosis of

kernel validity and preventing reward hacking that arises from naive syntax-only verification. By incorporating hierarchical reward assignment, token-level credit allocation, and strategic data mixing across SFT and RL stages, TritonRL stabilizes multi-turn training and yields improved kernel quality, generalization, and robustness. At the 8B scale, TritonRL surpasses prior Triton-specific models including KernelLLM [44] and AutoTriton[48], demonstrating how reinforcement learning can coordinate complex verification and generation workflows rather than merely improving individual code quality.

While prior training-based approaches primarily focus on dense kernels, **SparseRL** [53] extends reinforcement learning to sparsity-constrained CUDA kernel generation, where legality and performance are tightly coupled. Unlike general kernel optimization, sparse computing introduces hard structural constraints that must be respected throughout the optimization process, making reward design and exploration substantially more challenging. SparseRL directly fine-tunes a language model using RL to improve kernel correctness, sparsity-aware performance, and execution efficiency. The method formulates sparse kernel generation as a sequential decision process, where the model receives verifiable rewards based on compile success, functional correctness, sparsity legality, and runtime performance. Through repeated interaction with the execution environment, the model learns to apply domain specific sparse optimizations. Evaluated across a diverse suite of sparse CUDA kernels, SparseRL significantly outperforms supervised baselines and demonstrates strong generalization to unseen sparsity patterns. As a training-based RL approach, SparseRL highlights the effectiveness of reinforcement learning in enabling models to internalize complex hardware-aware optimization strategies for sparse GPU workloads.

Inspired by human staged optimization, Zhu et al. [54] propose the **MTMC** framework. MTMC separates the complex task into two coordinated components: *Macro Thinking*, which employs RL to train lightweight LLMs in efficiently exploring and learning semantic optimization strategies that maximize hardware utilization, and *Micro Coding*, which leverages general-purpose LLMs to incrementally implement stepwise optimization proposals. This decoupling allows the framework to navigate the vast optimization space while maintaining implementation correctness, avoiding the errors inherent in kernel generation. Evaluated on KernelBench [17], MTMC achieves near 100% and 70% accuracy at Levels 1-2 and 3, over 50% than SOTA general purpose and domain finetuned LLMs, with up to 7.3x speedup over LLMs, and 2.2x over expert optimized PyTorch Eager kernels. On the TritonBench [20], MTMC attains up to 59.64% accuracy and 34x speedup.

D. Challenges in LLM-Based Kernel Generation Methods

LLM-based kernel generation faces multiple key challenges across evaluation dimensions. For functional correctness and acceleration performance, agent-based methods have achieved steady improvements by leveraging iterative feedback from compilers or runtime profiling. However, these approaches come at the cost of substantial computational overhead and multiple iterations, raising efficiency concerns. Purely training-based methods are faster but often produce semantically flawed kernels, resulting in incorrect results, degraded performance, instability under edge-case inputs, or suboptimal optimization.

Robustness, versatility, cross-architecture portability, and reproducibility remain significant challenges for both agent-based and purely learning-based approaches. Models struggle to generalize across diverse operators, tensor shapes, data types, and hardware backends. The underlying cause is the scarcity of relevant training data and the models' insufficient ability to capture kernel semantics, including memory access patterns, parallel execution constraints, and data dependencies.

V. Roadmap for Advancing LLM-Based Kernel Generation

This section addresses **RQ3** by presenting a forward-looking roadmap for advancing LLM-based GPU kernel generation. Specifically, we identify two directions: integrating accumulated human expert knowledge and adapting ongoing technical advances in general-purpose code LLMs for kernel-specific generation. Human expertise in manual kernel optimization from the pre-LLM era encapsulates extensive domain knowledge, performance heuristics, and hardware-aware optimization strategies. Current LLMs, however, are unable to fully capture this expertise through purely data-driven training. Consequently, incorporating such human knowledge in an agentic way into the LLMs has the potential to substantially improve kernel generation. Moreover, although recent advances in general-purpose code models have led to substantial improvements in overall coding capabilities, these models are not specifically adapted for kernel-level optimization, leaving considerable room to adapt and extend such advances to kernel-specific generation and optimization. In the following, we accordingly organize the roadmap into two subsections.

A. Integrating Human Expertise

This subsection provides a systematic summary of prior human expertise that can inform performance optimization in LLMs. In particular, we emphasize key perspectives of human expertise, including mathematical equivalence transformations, data locality optimization, hardware instruction mapping, dynamic-to-static transformations, and precision–resource trade-offs. We illustrate these perspectives with a 2×2 convolution on a 3×3 input along the PyTorch-to-GPU execution pathway.

1) Mathematical Equivalence Transformations: Mathematical equivalence transformations reformulate computational problems into mathematically equivalent forms that can be implemented more efficiently.

We illustrate this principle using the im2col transformation as an example. As showed in Figure 4, with a 3×3 input and a 2×2 kernel, im2col flattens each overlapping patch into a column and concatenates them into a matrix—converting convolution from scattered dot products into a single dense matrix multiplication (GEMM). This restructuring enables the use of optimized GEMM libraries and hardware accelerators.

Table IV
WORKS AND PUBLICATION DATES WITH OPEN SOURCE STATUS

LLM-based Code Generation Work	Date	Type	Open Source?	Benchmarking
KernelBench [17]	2025.02	Single-Agent	✓	KernelBench
Chen et al. [26]	2025.02	Single-Agent	×	KernelBench
CuAsmRL [11]	2025.03	Training-based	×	Others
Brabec et al. [27]	2025.04	Single-Agent	✓	Others
KernelLLM [44]	2025.05	Training-based	✓	KernelBench-Triton[2]
CUDA-LLM [28]	2025.06	Single-Agent	×	KernelBench, CUDA Samples [55], Leet-GPU [56]
GPU Kernel Scientist [31]	2025.06	Multi-Agent	×	Others
Kevin [46]	2025.07	Training-based	×	KernelBench
CUDA-L1 [47]	2025.07	Training-based	✓	KernelBench
Geak [32]	2025.07	Multi-Agent	✓	TritonBench-revised Benchmark[3]
AutoTriton [48]	2025.07	Training-based	✓	KernelBench, TritonBench
Mishra and Nangia [33]	2025.07	Multi-Agent	×	Others
SwizzlePerf [12]	2025.08	Training-based	×	Others
Hao et al. [4]	2025.09	Mobile System	✓	Others
Astra [34]	2025.09	Multi-Agent	✓	Others
AI CUDA Engineer [19]	2025.09	Multi-Agent	×	robust-kbench
ConCuR [50]	2025.10	Training-based	✓	KernelBench
EVOENGINEER [29]	2025.10	Single-Agent	✓	Others
STARK [36]	2025.10	Multi-Agent	×	KernelBench
Nichols et al. [51]	2025.10	Training-based	×	KernelBench
TRITONRL [52]	2025.10	Training-based	✓	KernelBench
KernelFalcon [37]	2025.11	Multi-Agent	✓	KernelBench
CudaForge [38]	2025.11	Multi-Agent	✓	KernelBench (sampled)
PRAGMA [39]	2025.11	Multi-Agent	×	KernelBench
SparseRL [53]	2025.11	Training-based	×	Others
KERNELBAND [30]	2025.11	Single-Agent	×	TritonBench
MTMC [54]	2025.11	Training-based	×	KernelBench, TritonBench
KForge [41]	2025.11	Multi-Agent	×	Kernelbench
PIKE [42]	2025.11	Multi-Agent	×	METR-refined [57] variant of KernelBench
TritonForge [40]	2025.12	Multi-Agent	×	TritonBench
AKG [43]	2025	Multi-Agent	✓	KernelBench

[2] KernelBench-Triton is a variant of KernelBench [17], adapted specifically for evaluating Triton kernel generation.
[3] TritonBench-revised is an enhanced version of TritonBench, where Wang et al. [32] corrected kernel errors and fixed missing function calls in the original evaluation suite.

Notes: ✓ = Open Source, × = Not Open Source.
Type: *Training-based*, *Agent*, *Multi-Agent*, *Dataset*, *Mobile System*.
Benchmarking: Named benchmarks are used as indicated; "Others" refers to custom or unspecified evaluation suites; "-" indicates no benchmarking was declared or applicable.

2) Data Locality Optimization: Data locality optimization targets the fundamental memory bandwidth bottleneck in GPU computing. This perspective maximizes utilization of the GPU memory hierarchy from global memory through shared memory to registers by strategically restructuring data placement and access patterns to minimize data movement and maximize reuse. Specifically, coalesced memory access allows threads to load contiguous data efficiently. Tiled shared memory and kernel weight reuse in registers, combined with cache-aware layouts and bank-conflict avoidance, can also provide significant benefits [58]. Specifically, data locality optimization covers the following dimensions.

Coalesced Memory Access. Coalescing ensures that consecutive threads within a warp access consecutive memory locations, enabling a single wide memory transaction (e.g., 128 bytes) to serve multiple threads efficiently. For regular access patterns, this can be achieved through proper data layout, thread organization, on-chip memory utilization, and techniques such as reorganizing threads [59, 60], selecting optimal thread block sizes [9], transforming data layouts (e.g., array of structs to struct of arrays), and tiling [61–63]. For irregular access patterns, such as those in sparse matrices, specialized data formats are required to maintain coalesced memory access [64].

Shared Memory Tiling. Tiling (or spatial blocking) partitions data into blocks that fit within a streaming multiprocessor's shared memory, enabling high-bandwidth data reuse across multiple computations. This technique is particularly effective for operations with regular access patterns such as matrix multiplication and convolution. Shared memory tiling exploits both temporal locality (reusing data across multiple operations) [65–67] and spatial locality (accessing nearby data) [68–70].

Kernel Fusion. Kernel fusion merges multiple consecutive kernels into a single kernel, eliminating intermediate global memory writes and reads. This optimization reduces both memory traffic and kernel launch overhead. Key benefits include improved data reuse and enhanced cache utilization [71].

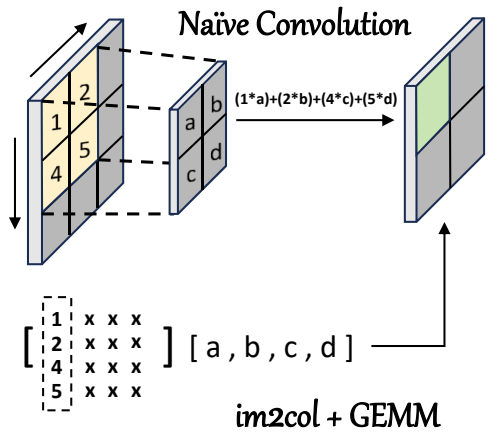

Figure 4. Schematic comparison of native convolution versus im2col+GEMM convolution. Up: Native convolution performs a sliding-window dot product: the kernel $[a, b, c, d]$ convolves with the input patch $[1, 2, 4, 5]$ to produce one output element $(1 \cdot a) + (2 \cdot b) + (4 \cdot c) + (5 \cdot d)$. Down: im2col+GEMM first flattens all sliding windows into columns of a matrix (only the first column $[1, 2, 4, 5]^T$ is shown), then multiplies the flattened kernel $[a, b, c, d]$ with this matrix in a single GEMM operation, producing all output elements simultaneously.

However, fusion may increase register and shared memory pressure, requiring careful trade-off analysis [72].

Register Blocking. Registers provide the fastest memory tier, with zero access latency. Register blocking (or temporal blocking) stores frequently reused values (e.g., kernel weights in convolution) in registers throughout computation; therefore this technique is especially effective for algorithms with high temporal locality [73, 74].

Prefetching. To reduce long memory latencies, data prefetching loads data for future computation steps before they are needed, overlapping memory transfers with computation. This technique is commonly applied in dense linear algebra kernels (e.g., matrix multiplication [75]) and stencil operations [66], often in combination with tiling and double buffering [76].

3) Hardware Instruction Optimization: Hardware instruction optimization focuses on mapping computation to efficient GPU instructions and scheduling them to maximize execution-unit utilization. At this level, the algorithmic structure and data layout are largely fixed, and performance improvements are achieved by exploiting instruction-level parallelism and hardware-specific execution characteristics. Specifically, optimization must address two fundamental tensions: ❶ the severe mismatch between peak arithmetic throughput and long memory and instruction latencies, and ❷ the contention for limited on-chip resources (e.g., registers and shared memory) induced by massive thread-level parallelism. Effective instruction-level optimizations therefore require careful coordination of instruction selection, scheduling, and resource allocation to fully exploit the underlying GPU microarchitecture.

Considerable human effort has historically gone into addressing the two fundamental performance tensions in GPU kernels. To mitigate the mismatch between high arithmetic throughput and long memory and instruction latencies, expert developers maximized parallelism at multiple levels. At the instruction level, they applied loop unrolling and instruction scheduling to increase instruction-level parallelism (ILP) and keep deep CUDA core pipelines occupied. Vectorization (e.g., using float4) further enhanced throughput by enabling SIMD execution and improving memory coalescing [77, 78]. At the thread-group level, warp-centric programming and warp shuffle operations facilitated efficient data exchange and reduction without shared-memory synchronization, sustaining warp activity and improving latency hiding [79, 80].

Similarly, managing contention for limited on-chip resources required careful tuning of work granularity and execution mapping. Thread coarsening illustrated a key trade-off: assigning more computation per thread improved ILP and register-level data reuse but increased register pressure, potentially reducing occupancy and limiting latency hiding [81]. Likewise, offloading computation to specialized execution units such as Tensor Cores achieved orders-of-magnitude higher throughput for mixed-precision GEMM [82], but imposed strict constraints on data layout, problem size, and kernel design, influencing overall resource allocation and scheduling decisions.

Previous work has shown that reducing numerical precision can improve computational efficiency and resource utilization, offering strategies that may also benefit large-scale model training and deployment. To balance computation, memory usage, and accuracy, practitioners selectively reduced precision, using FP16 and INT8 operations to leverage specialized hardware while maintaining acceptable model accuracy.

During training, techniques such as Automatic Mixed Precision (AMP) dynamically combine 16-bit and 32-bit operations, achieving significant speedups and memory savings without compromising accuracy [83]. For deployment, aggressive post-training quantization maps weights and activations to low-bit representations (e.g., INT4), reducing model size and enabling execution on resource-constrained devices, while quantization-aware training mitigates accuracy loss by embedding rounding and clipping directly into the forward pass [84]. Beyond precision alone, co-designing data layouts with reduced-precision arithmetic—such as transforming from Array-of-Structs to Struct-of-Arrays—has been shown to unlock additional speedups on GPUs by improving memory access patterns and cache utilization [85].

Current LLMs fall short of fully understanding human domain expertise. Integrating such expertise in an agentic way could substantially improve LLM-driven kernel generation.

B. Adapting Technical Advances in General-Purpose Code LLMs

In this section, we distill recent advances in the general code domain into a set of high-level principles (P1–P3) that may guide the advancement of DL kernel generation.

• **P1: Execution Semantic Integration.** Most LLM-based code generation approaches predominantly operate on textual

program representations, implicitly treating source code as a sufficient proxy for execution semantics relevant to optimization. However, in general programming tasks, it has been shown that textual representations alone are often insufficient to ensure functional correctness.

To address this limitation, some studies have explored ways to embed richer code semantics into model training, enabling LLMs to better capture execution-level behaviors and performance-relevant properties [86, 87]. This suggests that, for kernel generation, explicitly incorporating semantic information such as execution-level behaviors from kernel code into LLM-based methods could substantially enhance their ability to generate functionally correct kernels.

• **P2: Performance-Aware Semantic Integration.** Performance-oriented code generation, such as kernel optimization, differs from standard code generation in that it requires balancing runtime efficiency and functional correctness, which could be framed as a multi-objective optimization problem. Recent advances [88–90] have been proposed to address multi-objective optimization in the general-purpose code domain. These developments also suggest a path forward for LLM-based kernel generation.

• **P3: Hardware-Aware Cost Modeling.** Kernel optimization is inherently hardware-aware: performance depends critically on factors such as memory access patterns, cache behavior, parallelism, and synchronization. Current LLMs, however, lack native awareness of these hardware constraints, limiting their ability to reliably generate high-performance kernels. Recent advances [91] have focused on predicting numeric outcomes of code executions, which has the potential to be applied in hardware-aware cost modeling. This could be further leveraged to improve hardware-aware kernel generation in LLMs.

VI. Conclusion

This survey provides an overview of current benchmarks and techniques for LLM-driven kernel generation and optimization. Despite notable progress, systematically improving the performance of existing methods remains challenging. We summarize insights from the pre-LLM era and the broader code generation domain that may inform future advances in kernel optimization. We hope this survey serves as a timely reference and motivates further research on DL kernel generation and optimization.

References

[1] J. Sevilla, L. Heim, A. Ho, T. Besiroglu, M. Hobbhahn, and P. Villalobos, "Compute trends across three eras of machine learning," in *2022 international joint conference on neural networks (IJCNN)*. IEEE, 2022, pp. 1–8.

[2] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[3] S. Kalade and G. Schelle, "Npueval: Optimizing npu kernels with llms and open source compilers," *arXiv preprint arXiv:2507.14403*, 2025.

[4] Z. Hao, J. Wei, T. Wang, M. Huang, H. Jiang, S. Jiang, T. Cao, and J. Ren, "Scaling llm test-time compute with mobile npu on smartphones," *arXiv preprint arXiv:2509.23324*, 2025.

[5] M.-K. Nguyen-Nhat, H. D. N. Do, H. T. Le, and T. T. Dao, "Llmperf: Gpu performance modeling meets large language models," in *2024 32nd International Conference on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2024, pp. 1–8.

[6] G. Bolet, G. Georgakoudis, H. Menon, K. Parasyris, N. Hasabnis, H. Estes, K. Cameron, and G. Oren, "Can large language models predict parallel code performance?" in *Proceedings of the 34th International Symposium on High-Performance Parallel and Distributed Computing*, 2025, pp. 1–6.

[7] C. Kachris, "A survey on hardware accelerators for large language models," *Applied Sciences*, vol. 15, no. 2, p. 586, 2025.

[8] S. Hong and H. 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*, 2009, pp. 152–163.

[9] A. Leist, D. P. Playne, and K. A. Hawick, "Exploiting graphical processing units for data-parallel scientific applications," *Concurrency and Computation: Practice and Experience*, vol. 21, no. 18, pp. 2400–2437, 2009.

[10] B. F. Spector, S. Arora, A. Singhal, D. Y. Fu, and C. Ré, "Thunderkittens: Simple, fast, and adorable ai kernels," *arXiv preprint arXiv:2410.20399*, 2024.

[11] G. He and E. Yoneki, "Cuasmrl: Optimizing gpu sass schedules via deep reinforcement learning," in *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, 2025, pp. 493–506.

[12] A. Tschand, M. Awad, R. Swann, K. Ramakrishnan, J. Ma, K. Lowery, G. Dasika, and V. J. Reddi, "Swizzleperf: Hardware-aware llms for gpu kernel performance optimization," *arXiv preprint arXiv:2508.20258*, 2025.

[13] P. Hijma, S. Heldens, A. Sclocco, B. Van Werkhoven, and H. E. Bal, "Optimization techniques for gpu programming," *ACM Computing Surveys*, vol. 55, no. 11, pp. 1–81, 2023.

[14] A. Santa Molison, M. Moraes, G. Melo, F. Santos, and W. K. Assuncao, "Is llm-generated code more maintainable\& reliable than human-written code," *arXiv preprint arXiv:2508.00700*, 2025.

[15] B. Varghese, N. Wang, D. Bermbach, C.-H. Hong, E. D. Lara, W. Shi, and C. Stewart, "A survey on edge performance benchmarking," *ACM Computing Surveys (CSUR)*, vol. 54, no. 3, pp. 1–33, 2021.

[16] Triton Development Community, "Triton documentation," https://triton-lang.org/main/index.html, 2026, accessed: 2026-01. [Online]. Available: https://triton-lang.org/main/index.html

[17] A. Ouyang, S. Guo, S. Arora, A. L. Zhang, W. Hu, C. Ré, and A. Mirhoseini, "Kernelbench: Can llms write efficient gpu kernels?" *arXiv preprint arXiv:2502.10517*, 2025.

[18] Z. Wen, Y. Zhang, Z. Li, Z. Liu, L. Xie, and T. Zhang, "Multikernelbench: A multi-platform benchmark for kernel generation," *arXiv e-prints*, pp. arXiv–2507, 2025.

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

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

[21] Meta, "Backendbench," [Online]. Available: https://github.com/meta-pytorch/BackendBench, 2025, accessed: Dec. 21, 2025.

[22] NVIDIA, "compute-eval," [Online]. Available: https://github.com/NVIDIA/compute-eval, 2025, accessed: Dec. 21, 2025.

[23] OpenAI, "Gpt-4o system card," 2024. [Online]. Available: https://arxiv.org/abs/2410.21276

[24] Deepseek-AI, "Deepseek-r1 incentivizes reasoning in llms through reinforcement learning," *Nature*, vol. 645, no. 8081, p. 633–638, Sep. 2025. [Online]. Available: http://dx.doi.org/10.1038/s41586-025-09422-z

[25] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," 2023. [Online]. Available: https://arxiv.org/abs/2302.13971

[26] T. Chen, B. Xu, and K. Devleker, "Automating gpu kernel generation with deepseek-r1 and inference time scaling," Online: NVIDIA Blog, https://developer.nvidia.com/blog/automating-gpu-kernel-generation-with-deepseek-r1-and-inference-time-scaling/, 2025, accessed: Nov. 11, 2025.

[27] M. Brabec, J. Klepl, M. Töpfer, and M. Kruliš, "Tutoring llm into a better cuda optimizer," in *European Conference on Parallel Processing*. Springer, 2025, pp. 250–263.

[28] W. Chen, J. Zhu, Q. Fan, Y. Ma, and A. Zou, "Cuda-llm: Llms can write efficient cuda kernels," *arXiv preprint arXiv:2506.09092*, 2025.

[29] P. Guo, C. Zhu, S. Chen, F. Liu, X. Lin, Z. Lu, and Q. Zhang, "Evoengineer: Mastering automated cuda kernel code evolution with large language models," *arXiv preprint arXiv:2510.03760*, 2025.

[30] D. Ran, S. Xie, M. Ji, Z. Hua, M. Wu, Y. Cao, Y. Guo, Y. Hao, L. Li, Y. Hu *et al.*, "Kernelband: Boosting llm-based kernel optimization with a hierarchical and hardware-aware multi-armed bandit," *arXiv preprint arXiv:2511.18868*, 2025.

[31] M. Andrews and S. Witteveen, "Gpu kernel scientist: An llm-driven framework for iterative kernel optimization," *arXiv preprint arXiv:2506.20807*, 2025.

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

[33] S. Mishra and A. Nangia, "How many agents does it take to beat pytorch? (surprisingly not that much)," https://letters.lossfunk.com/p/how-many-agents-does-it-take-to-beat, Jul. 2025, accessed: 2025-11-12.

[34] A. Wei, T. Sun, Y. Seenichamy, H. Song, A. Ouyang, A. Mirhoseini, K. Wang, and A. Aiken, "Astra: A multi-agent system for gpu kernel performance optimization," *arXiv preprint arXiv:2509.07506*, 2025.

[35] L. Zheng, L. Yin, Z. Xie, C. L. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez *et al.*, "Sglang: Efficient execution of structured language model programs," *Advances in neural information processing systems*, vol. 37, pp. 62 557–62 583, 2024.

[36] J. Dong, Y. Yang, T. Liu, Y. Wang, F. Qi, V. Tarokh, K. Rangadurai, and S. Yang, "Stark: Strategic team of agents for refining kernels," *arXiv preprint arXiv:2510.16996*, 2025.

[37] L. Wang and the PyTorch Team at Meta, "Kernelfalcon: Autonomous gpu kernel generation via deep agents," [Online]. Available: https://pytorch.org/blog/kernelfalcon-autonomous-gpu-kernel-generation-via-deep-agents/, 2025, accessed: Nov. 11, 2025.

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

[39] K. Lei, H. Yang, H. Zhang, X. You, K. Zhang, Z. Luan, Y. Liu, and D. Qian, "Pragma: A profiling-reasoned multi-agent framework for automatic kernel optimization," *arXiv preprint arXiv:2511.06345*, 2025.

[40] H. Li, K. Man, P. Kanuparthy, H. Chen, W. Sun, S. Tallam, C. Zhu, K. Zhu, and Z. Qian, "Tritonforge: Profiling-guided framework for automated triton kernel optimization," *arXiv preprint arXiv:2512.09196*, 2025.

[41] T. Sereda, T. S. John, B. Bartan, N. Serrino, S. Katti, and Z. Asgar, "Kforge: Program synthesis for diverse ai hardware accelerators," *arXiv preprint arXiv:2511.13274*, 2025.

[42] K. Nagaitsev, L. Grbcic, S. Williams, and C. Iancu, "Optimizing pytorch inference with llm-based multi-agent systems," *arXiv preprint arXiv:2511.16964*, 2025.

[43] MindSpore, "Auto kernel generator (akg)," [Online]. Available: https://atomgit.com/mindspore/akg, 2025, accessed: Dec. 18, 2025.

[44] Z. V. Fisches, S. Paliskara, S. Guo, A. Zhang, J. Spisak, C. Cummins, H. Leather, G. Synnaeve, J. Isaacson, A. Markosyan, and M. Saroufim, "Kernelllm: Making kernel development more accessible," [Online]. Available: https://huggingface.co/facebook/KernelLLM, 2025, accessed: Nov. 11, 2025.

[45] D. Kocetkov, R. Li, L. B. Allal, J. Li, C. Mou, C. M. Ferrandis, Y. Jernite, M. Mitchell, S. Hughes, T. Wolf *et al.*, "The stack: 3 tb of permissively licensed source

code," *arXiv preprint arXiv:2211.15533*, 2022.

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

[47] X. Li, X. Sun, A. Wang, J. Li, and C. Shum, "Cuda-l1: Improving cuda optimization via contrastive reinforcement learning," *arXiv preprint arXiv:2507.14111*, 2025.

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

[49] Z. Shao, P. Wang, Q. Zhu, R. Xu, J. Song, X. Bi, H. Zhang, M. Zhang, Y. Li, Y. Wu *et al.*, "Deepseekmath: Pushing the limits of mathematical reasoning in open language models," *arXiv preprint arXiv:2402.03300*, 2024.

[50] L. Kong, J. Wei, H. Shen, and H. Wang, "Concur: Conciseness makes state-of-the-art kernel generation," *arXiv preprint arXiv:2510.07356*, 2025.

[51] D. Nichols, K. Parasyris, C. Jekel, A. Bhatele, and H. Menon, "Integrating performance tools in model reasoning for gpu kernel optimization," *arXiv preprint arXiv:2510.17158*, 2025.

[52] J. Woo, S. Zhu, A. Nie, Z. Jia, Y. Wang, and Y. Park, "Tritonrl: Training llms to think and code triton without cheating," *arXiv preprint arXiv:2510.17891*, 2025.

[53] Anonymous, "Mastering sparse CUDA generation through pretrained models and deep reinforcement learning," in *Submitted to The Fourteenth International Conference on Learning Representations*, 2025, under review. [Online]. Available: https://openreview.net/forum?id=VdLEaGPYWT

[54] X. Zhu, S. Peng, J. Guo, Y. Chen, Q. Guo, Y. Wen, H. Qin, R. Chen, Q. Zhou, K. Gao *et al.*, "Qimeng-kernel: Macro-thinking micro-coding paradigm for llm-based high-performance gpu kernel generation," *arXiv preprint arXiv:2511.20100*, 2025.

[55] NVIDIA Corporation, "Cuda code samples," [Online]. Available: https://github.com/NVIDIA/cuda-samples, 2025, accessed: Dec. 18, 2025.

[56] LeetGPU, "Challenges," [Online]. Available: https://leetgpu.com/challenges, 2025, accessed: Dec. 18, 2025.

[57] METR, "Measuring automated kernel engineering," [Online]. Available: https://metr.org/blog/2025-02-14-measuring-automated-kernel-engineering/, 2025, accessed: Dec. 18, 2025.

[58] P. Hijma, S. Heldens, A. Sclocco, B. Van Werkhoven, and H. E. Bal, "Optimization techniques for gpu programming," *ACM Computing Surveys*, vol. 55, no. 11, pp. 1–81, 2023.

[59] J. Gómez-Luna, J. M. González-Linares, J. I. Benavides, and N. Guil, "An optimized approach to histogram computation on gpu," *Machine Vision and Applications*, vol. 24, no. 5, pp. 899–908, 2013.

[60] Y. Hu, H. Liu, and H. H. Huang, "Tricore: Parallel triangle counting on gpus," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 171–182.

[61] R. Nath, S. Tomov, T. T. Dong, and J. Dongarra, "Optimizing symmetric dense matrix-vector multiplication on gpus," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–10.

[62] W. T. Tang, W. J. Tan, R. Ray, Y. W. Wong, W. Chen, S.-h. Kuo, R. S. M. Goh, S. J. Turner, and W.-F. Wong, "Accelerating sparse matrix-vector multiplication on gpus using bit-representation-optimized schemes," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12.

[63] R. Nath, S. Tomov, and J. Dongarra, "An improved magma gemm for fermi graphics processing units," *The International Journal of High Performance Computing Applications*, vol. 24, no. 4, pp. 511–515, 2010.

[64] J. Zhong and B. He, "Medusa: Simplified graph processing on gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1543–1552, 2013.

[65] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra, "Performance tuning and optimization techniques of fixed and variable size batched cholesky factorization on gpus," *Procedia Computer Science*, vol. 80, pp. 119–130, 2016.

[66] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on gpu architectures," in *Proceedings of the 26th ACM international conference on Supercomputing*, 2012, pp. 311–320.

[67] Y. Do, H. Kim, P. Oh, D. Park, and J. Lee, "Snu-npb 2019: parallelizing and optimizing npb in opencl and cuda for modern gpus," in *2019 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2019, pp. 93–105.

[68] M. Fang, J. Fang, W. Zhang, H. Zhou, J. Liao, and Y. Wang, "Benchmarking the gpu memory at the warp level," *Parallel Computing*, vol. 71, pp. 23–41, 2018.

[69] F. Petrovič, D. Střelák, J. Hozzová, J. Ol'ha, R. Trembeckỳ, S. Benkner, and J. Filipovič, "A benchmark set of highly-efficient cuda and opencl kernels and its dynamic autotuning with kernel tuning toolkit," *Future Generation Computer Systems*, vol. 108, pp. 161–177, 2020.

[70] J. Huang, C. D. Yu, and R. A. v. d. Geijn, "Strassen's algorithm reloaded on gpus," *ACM Transactions on Mathematical Software (TOMS)*, vol. 46, no. 1, pp. 1–22, 2020.

[71] M. Korch and T. Werner, "Accelerating explicit ode methods on gpus by kernel fusion," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 18, p. e4470, 2018.

[72] J. Carabaño, J. Westerholm, and T. Sarjakoski, "A compiler approach to map algebra: automatic parallelization, locality optimization, and gpu acceleration of raster spatial analysis," *GeoInformatica*, vol. 22, no. 2, pp. 211–235, 2018.

[73] N.-P. Tran, M. Lee, and D. H. Choi, "Memory-efficient parallelization of 3d lattice boltzmann flow solver on a gpu," in *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*. IEEE, 2015, pp.

315–324.

[74] W. Qiu, Z. Gong, Y. Guo, B. Liu, X. Tang, and Y. Yuan, "Gpu-based high performance password recovery technique for hash functions." *J. Inf. Sci. Eng.*, vol. 32, no. 1, pp. 97–112, 2016.

[75] P. N. Q. Anh, R. Fan, and Y. Wen, "Balanced hashing and efficient gpu sparse general matrix-matrix multiplication," in *Proceedings of the 2016 International Conference on Supercomputing*, 2016, pp. 1–12.

[76] K. Matam, S. R. K. B. Indarapu, and K. Kothapalli, "Sparse matrix-matrix multiplication on modern architectures," in *2012 19th International Conference on High Performance Computing*. IEEE, 2012, pp. 1–10.

[77] A. A. Abdelrahman, M. M. Fouad, H. Dahshan, and A. M. Mousa, "High performance cuda aes implementation: A quantitative performance analysis approach," in *2017 Computing conference*. IEEE, 2017, pp. 1077–1085.

[78] G.-J. van den Braak, B. Mesman, and H. Corporaal, "Compile-time gpu memory access optimizations," in *2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*. IEEE, 2010, pp. 200–207.

[79] M. Bauer, H. Cook, and B. Khailany, "Cudadma: optimizing gpu memory bandwidth via warp specialization," in *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, 2011, pp. 1–11.

[80] T. Honda, Y. Ito, and K. Nakano, "A warp-synchronous implementation for multiple-length multiplication on the gpu," in *2015 Third International Symposium on Computing and Networking (CANDAR)*. IEEE, 2015, pp. 96–102.

[81] J. D. Garvey and T. S. Abdelrahman, "A strategy for automatic performance tuning of stencil computations on gpus," *Scientific Programming*, vol. 2018, no. 1, p. 6093054, 2018.

[82] D. Yan, W. Wang, and X. Chu, "Demystifying tensor cores to optimize half-precision matrix multiply," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 634–643.

[83] M. M. H. Opi, S. Khan, and M. F. Rahman, "Accelerating bangla nlp tasks with automatic mixed precision: Resource-efficient training preserving model efficacy," *arXiv preprint arXiv:2512.00829*, 2025.

[84] E. Nyamsuren, "Evaluating quantized large language models for code generation on low-resource language benchmarks," *arXiv preprint arXiv:2410.14766*, 2024.

[85] P. K. Radtke and T. Weinzierl, "Compiler-supported reduced precision and aos-soa transformations for heterogeneous hardware," *arXiv preprint arXiv:2512.05516*, 2025.

[86] J. Li, D. Guo, D. Yang, R. Xu, Y. Wu, and J. He, "Codei/o: Condensing reasoning patterns via code input-output prediction," *arXiv preprint arXiv:2502.07316*, 2025.

[87] Y. Ding, J. Peng, M. Min, G. Kaiser, J. Yang, and B. Ray, "Semcoder: Training code language models with comprehensive semantics reasoning," *Advances in Neural Information Processing Systems*, vol. 37, pp. 60 275–60 308, 2024.

[88] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," *arXiv preprint arXiv:2406.00515*, 2024.

[89] M. Du, L. A. Tuan, Y. Liu, Y. Qing, D. Huang, X. He, Q. Liu, Z. Ma, and S.-k. Ng, "Afterburner: Reinforcement learning facilitates self-improving code efficiency optimization," *arXiv preprint arXiv:2505.23387*, 2025.

[90] J. Yang, X. Liu, W. Lv, K. Deng, S. Guo, L. Jing, Y. Li, S. Liu, X. Luo, Y. Luo *et al.*, "From code foundation models to agents and applications: A comprehensive survey and practical guide to code intelligence," *arXiv preprint arXiv:2511.18538*, 2025.

[91] Y. Akhauri, X. Song, A. Wongpanich, B. Lewandowski, and M. S. Abdelfattah, "Regression language models for code," *arXiv preprint arXiv:2509.26476*, 2025.