

# Recent Advances in Deep Learning Kernel Optimization Using Large Language Models

TIANLIN LI\*, Beihang University, China  
CHENXI YANG\*, Tianjin University, China  
QITONG SUN, Beihang University, China  
XIAOYU ZHANG, Nanyang Technological University, Singapore  
QIANG HU, Tianjin University, China  
ZHE TANG, Zhejiang Lab, China  
SHENG CHEN, Zhejiang Lab, China  
FEI YANG, Zhejiang Lab, China  
AISHAN LIU, Beihang University, China  
XIANGLONG LIU, Beihang University, China  
CHAO SHEN, Xi'an Jiaotong University, China  
YVES LE TRAON, University of Luxembourg, Luxembourg  
YANG LIU, Nanyang Technological University, Singapore

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.

Additional Key Words and Phrases: Large Language Models (LLMs), Deep Learning Kernels, Kernel Optimization, GPU Acceleration, Benchmarking

## 1 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

\*Equal contribution.

Authors' Contact Information: Tianlin Li, tianlin001@buaa.edu.cn, Beihang University, Beijing, China; Chenxi Yang, 3023244212@tju.edu.cn, Tianjin University, Tianjin, China; Qitong Sun, sunqt@buaa.edu.cn, Beihang University, Beijing, China; Xiaoyu Zhang, xiaoyu.zhang@ntu.edu.sg, Nanyang Technological University, Singapore, Singapore; Qiang Hu, qianghu@tju.edu.cn, Tianjin University, Tianjin, China; Zhe Tang, tangzhe@zhejianglab.org, Zhejiang Lab, Hangzhou, China; Sheng Chen, scucs@zhejianglab.org, Zhejiang Lab, Hangzhou, China; Fei Yang, yangf@zhejianglab.org, Zhejiang Lab, Hangzhou, China; Aishan Liu, liuaishan@buaa.edu.cn, Beihang University, Beijing, China; Xianglong Liu, xliliu@buaa.edu.cn, Beihang University, Beijing, China; Chao Shen, chaoshen@mail.xjtu.edu.cn, Xi'an Jiaotong University, Xi'an, China; Yves Le Traon, yves.letraon@uni.lu, University of Luxembourg, Luxembourg City, Luxembourg; Yang Liu, yangliu@ntu.edu.sg, Nanyang Technological University, Singapore, Singapore.

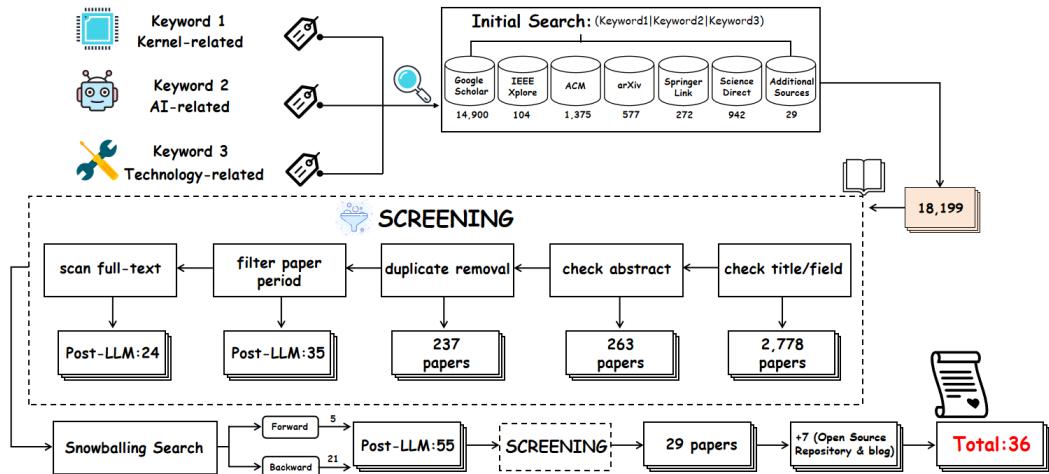


Fig. 1. The workflow and methodology of this survey.

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<sup>1</sup>. 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.

<sup>1</sup>For simplicity, we do not distinguish between kernel generation and optimization, and use the two terms interchangeably.

## 99      1.1 Survey Method

100     The survey method adopted for preparing this paper is based on an approach widely presented in  
101    previous surveys [15]. This method includes ① defining the objectives of the survey, ② defining  
102    research questions, ③ selecting keywords for searching, and ④ identifying criteria for including or  
103    excluding research. These aspects are defined below.

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

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

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

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

109    **RQ1** How are LLM-based kernel optimization methods currently evaluated? (response in Sec-  
110    tion 3)

111    **RQ2** How do existing approaches perform kernel optimization using LLMs? (response in Sec-  
112    tion 4)

113    **RQ3** What roadmap can guide future research in LLM-based kernel optimization? (response in  
114    Section 5)

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

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

130    The rest of this survey is organized as follows. Section 3 reviews existing performance benchmark-  
131    ing methodologies. Section 4 distinguishes between three major kernel optimization methodologies:  
132    single-agent systems, multi-agent systems, and training-based approaches, and organizes the sur-  
133   veyed methods accordingly. Section 5 outlines a forward-looking research roadmap, drawing on  
134    key insights from manual kernel optimization efforts in pre-LLM research and advances in the  
135    general code generation domain.

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

## 139      2 Background and Preliminaries

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

Table 1. Cross-vendor Mapping of GPU Architectural Terminology

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

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

## 2.1 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 1), such as 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.

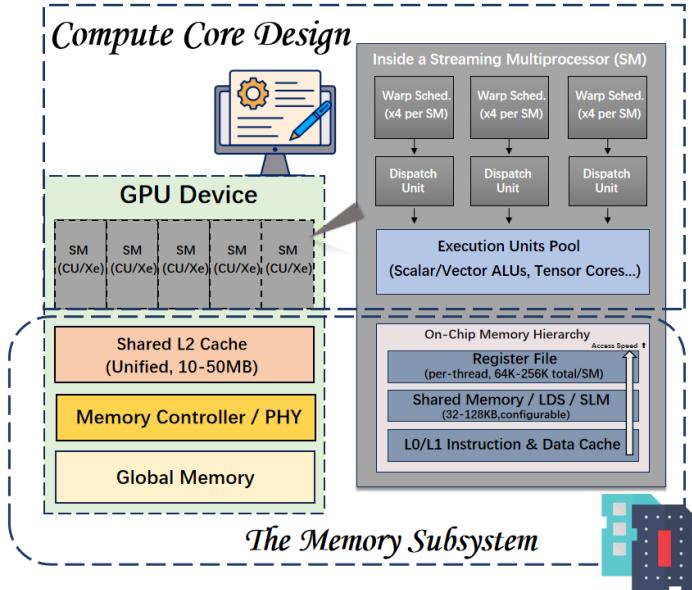


Fig. 2. GPU architecture overview. The diagram illustrates the two primary subsystems of a modern GPU. Up: Compute Core Design — The GPU comprises an array of parallel Streaming Multiprocessors (SMs, also called Compute Units or Xe-cores) that access off-chip Global Memory via a shared, unified L2 Cache 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.

## 2.2 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: ① Python API Invocation, where the user's operator call is captured; ② C++ ATen Dispatching, which dynamically selects the appropriate backend kernel; ③ CUDA Kernel Launch, where execution commands are prepared for the GPU; ④ 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.

**2.2.1 Python API Invocation.** First, developers write Python scripts that call PyTorch operators, for example:

```

251 1 import torch
252 2
253 3 # Initialize tensors
254 4 a = torch.tensor([1.0, 2.0], device='cuda')
255 5 b = torch.tensor([3.0, 4.0], device='cuda')
256 6
257 7 # Element-wise addition on GPU
258 8 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.

**2.2.2 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.

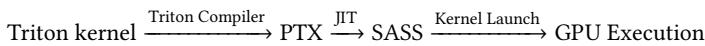
**2.2.3 Kernel Launch.** With the specific kernel function pointer, device pointers to tensor data, and calculated launch parameters from the previous stage, ATen calculates the optimal GPU thread configuration, including grid dimensions, thread-block dimensions, and other parameters, then asynchronously 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.

**2.2.4 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

295 maintaining low development overhead, Triton [16] was introduced as a high-level, Python-like  
 296 language specifically designed for GPU kernel development. Triton provides a more intuitive and  
 297 high-level programming interface, allowing developers to express complex GPU computation  
 298 patterns without writing low-level CUDA code. At the same time, it offers fine-grained control over  
 299 hardware resources, such as memory hierarchy, thread/block mapping, and vectorized operations,  
 300 giving expert users the ability to tune kernels for maximum performance. This combination of ease  
 301 of use and low-level control makes Triton a particularly promising tool for both rapid prototyping  
 302 and performance-critical kernel development.

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



### 318 2.3 Efficiency Bottlenecks of Kernels in DL Software Stacks

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

322 ❶ While PyTorch’s automation successfully abstracts hardware complexity and ensures functional  
 323 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  
 324 support a wide range of tensor shapes, data types, and hardware variants, which inherently prevents  
 325 them from exploiting the full performance potential of any specific workload. For example, a general  
 326 matrix multiplication kernel cannot assume fixed dimensions or memory layouts, and therefore  
 327 cannot apply aggressive, shape-specific optimizations such as tailored loop tiling, unrolling, or  
 328 memory-access patterns.

329 ❷ Moreover, PyTorch’s dynamic execution model defers key optimization decisions until runtime,  
 330 limiting the scope for deep static analysis and compilation. This constraint stems from its core design:  
 331 tensor properties such as shape, data type, and device are only known during execution, which  
 332 prevents the compiler from making aggressive, irreversible optimizations upfront. For instance,  
 333 while a matrix multiplication kernel generated at runtime must accommodate arbitrary tensor  
 334 shapes, a pre-optimized kernel can be specialized for a fixed size, such as  $256 \times 256$ . This specialization  
 335 enables compile-time optimizations unavailable in the dynamic path: explicit orchestration of data  
 336 movement across the memory hierarchy (global  $\rightarrow$  shared  $\rightarrow$  registers), fine-tuning of thread-block  
 337 and grid dimensions to maximize occupancy and latency hiding, and precise alignment of data  
 338 layouts and instruction sequences to exploit specialized hardware such as Tensor Cores or Matrix  
 339 Cores. Consequently, specialized kernels, whether manually engineered or auto-tuned, can achieve  
 340 substantially higher hardware utilization for their target workloads than the general-purpose  
 341

344 kernels dispatched through PyTorch’s default stack, often translating to measurable, multi-fold  
345 speedups across training and inference pipelines.

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

## 350 2.4 Problem Definition

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

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

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

364 Formally, DL kernel generation and optimization seek to solve:

$$\small 365 \quad K^* = \arg \max_{\substack{K \in \mathcal{G}(S, B), \\ \theta \in \mathcal{O}(K)}} P(K, \theta, H), \text{ such that } K \models S,$$

368 where:

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

## 376 2.5 Requirements for Kernel Optimization Methods

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

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

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

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

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

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

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

392

Table 2. Overview of Benchmarks for LLM-based GPU Kernel Generation

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

Table 3. Evaluation Dimensions Coverage Across GPU Kernel Generation Benchmarks

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

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

### 3 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 2 and Table 3. We then outline the challenges and present a roadmap.

#### 3.1 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. ① They remain confined to NVIDIA’s ecosystem (CUDA or Triton), leaving other hardware platforms unexamined. ② 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 [22] 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.

### 3.2 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 2.5, 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

491 overlook these conditions, leaving kernels vulnerable to failures or performance degradation in  
492 challenging situations.

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

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

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

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

### 511 512 **3.3 Roadmap for Advancing Benchmarking**

513 This roadmap outlines strategic directions to improve benchmark design.

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

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

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

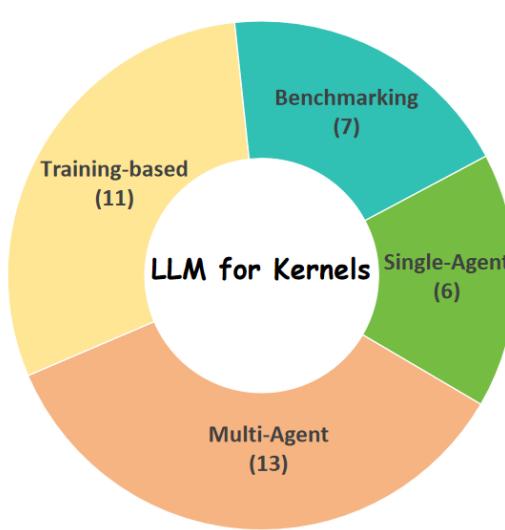


Fig. 3. Distribution of LLM-driven GPU kernel generation and optimization research. Percentages are calculated over the 37 surveyed post-LLM kernel generation and optimization papers. Works focusing primarily on mobile systems [4] are excluded from the statistics. KernelBench [17] and AI CUDA Engineer / 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.

## 4 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 4.

### 4.1 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 [52], DeepSeek-R1 [53], and Llama [54]. 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

Table 4. 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. [23]	2025.02	Single-Agent	✗	KernelBench
CuAsmRL [11]	2025.03	Training-based	✗	Others
Brabec et al. [24]	2025.04	Single-Agent	✓	Others
KernelLLM [25]	2025.05	Training-based	✓	KernelBench-Triton <sup>2</sup>
CUDA-LLM [26]	2025.06	Single-Agent	✗	KernelBench, CUDA Samples [27], LeetGPU [28]
GPU Kernel Scientist [29]	2025.06	Multi-Agent	✗	Others
Kevin [30]	2025.07	Training-based	✗	KernelBench
CUDA-L1 [31]	2025.07	Training-based	✓	KernelBench
Geak [32]	2025.07	Multi-Agent	✓	TritonBench-revised Benchmark <sup>3</sup>
AutoTriton [33]	2025.07	Training-based	✓	KernelBench, TritonBench
Mishra and Nangia [34]	2025.07	Multi-Agent	✗	Others
SwizzlePerf [12]	2025.08	Training-based	✗	Others
Hao et al. [4]	2025.09	Mobile System	✓	Others
Astra [35]	2025.09	Multi-Agent	✓	Others
AI CUDA Engineer [19]	2025.09	Multi-Agent	✗	robust-kbench
ConCuR [36]	2025.10	Training-based	✓	KernelBench
EVOENGINEER [37]	2025.10	Single-Agent	✓	Others
STARK [38]	2025.10	Multi-Agent	✗	KernelBench
Nichols et al. [39]	2025.10	Training-based	✗	KernelBench
TRITONRL [40]	2025.10	Training-based	✓	KernelBench
KernelFalcon [41]	2025.11	Multi-Agent	✓	KernelBench
CudaForge [42]	2025.11	Multi-Agent	✓	KernelBench (sampled)
PRAGMA [43]	2025.11	Multi-Agent	✗	KernelBench
SparseRL [44]	2025.11	Training-based	✗	Others
KERNELBAND [45]	2025.11	Single-Agent	✗	TritonBench
MTMC [46]	2025.11	Training-based	✗	KernelBench, TritonBench
KForge [47]	2025.11	Multi-Agent	✗	Kernelbench
PIKE [48]	2025.11	Multi-Agent	✗	METR-refined [49] variant of KernelBench
TritonForge [50]	2025.12	Multi-Agent	✗	TritonBench
AKG [51]	2025.12	Multi-Agent	✓	KernelBench

<sup>2</sup> KernelBench-Triton is a variant of KernelBench [17], adapted specifically for evaluating Triton kernel generation.<sup>3</sup> 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.

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 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. [23] 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. [24] 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. [26] 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. [23] 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 4.3.

Guo et al. [37] 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  $2.72\times$  over baseline CUDA kernels and a code validity rate of 69.8%, establishing a principled and reusable foundation for evolutionary kernel optimization.

From a decision theoretic perspective, Ran et al. [45] 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.

## 4.2 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 [29] 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 [34] 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.

736 Additionally, unlike prior systems that primarily generate optimized kernels from scratch, Wei  
 737 et al. [35] propose **Astra**, shifting the problem setting toward optimizing existing CUDA kernels  
 738 from SGLang [55], which is a widely deployed LLM serving framework. Astra organizes the op-  
 739 timization loop into four specialized agents. A *Testing Agent* constructs correctness test suites  
 740 and validates candidate kernels, while a *Profiling Agent* measures execution time and memory  
 741 behavior to provide hardware-level performance feedback. A *Planning Agent* jointly reasons over  
 742 correctness and profiling signals to propose targeted transformations, and a *Coding Agent* applies  
 743 these plans to synthesize new kernel implementations. To enable direct optimization of SGLang’s  
 744 highly interdependent kernels, Astra further introduces a pre-/post-processing pipeline that ex-  
 745 tracts kernels into stand-alone forms for optimization and subsequently reintegrates optimized  
 746 implementations back into the full framework for validation and benchmarking. This design allows  
 747 Astra to report speedups relative to the original production kernels while preserving compatibility  
 748 with the original framework. Evaluated on real SGLang kernels, Astra achieves consistent speedups  
 749 over single-agent baselines under zero-shot prompting, highlighting the practical potential of  
 750 multi-agent systems for maintaining and optimizing production GPU code.

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

762 Furthermore, **STARK** proposed by Dong et al. [38] advances prior LLM-based kernel optimizers  
 763 by redesigning kernel refinement as a tightly coordinated multi-agent process with strategic tree  
 764 search over persistent memory. STARK decomposes optimization into specialized planning, coding,  
 765 and debugging agents, and introduces grounded instructions and dynamic context windows to  
 766 translate high-level strategies into precise, localized CUDA code edits. Grounded instructions  
 767 anchor planned transformations to concrete code spans, specifying where and how to apply each  
 768 optimization, while dynamic context windows expose different historical attempts and feedback  
 769 to specific agents, enabling experience-guided planning, implementation, and debugging. This  
 770 design tightly couples strategic reasoning with low-level execution and balances exploration and  
 771 exploitation to systematically navigate the code space, mitigating common failure modes such  
 772 as incoherent refinements and myopic local search. Evaluated on KernelBench, STARK achieves  
 773 substantially higher success rates and runtime speedups (up to 10×–16×), particularly on kernels  
 774 where baseline agents struggle to produce valid implementations.

775 Meanwhile, Wang and the PyTorch Team at Meta [41] produce **KernelFalcon**, which organizes  
 776 kernel synthesis into a deterministic, orchestrated agent pipeline with decomposition, parallel  
 777 exploration, and execution-based verification. Its workflow is decomposed into specialized agents  
 778 responsible for operator fusion, subgraph extraction, Triton kernel synthesis, and end-to-end nu-  
 779 mercial validation, coordinated by a central Orchestrator that manages delegation, failure handling,  
 780 and early-stop parallel search. Crucially, KernelFalcon adopts a verifier-first loop: candidate kernels  
 781 are compiled and executed against PyTorch references, and the system early-exits upon discovering  
 782 numerically correct implementations, enabling parallel exploration of diverse kernel realizations  
 783 while preserving full PyTorch semantics. KernelFalcon is the first known open agentic system to

784

785 achieve 100% correctness across all 250 L1/L2/L3 KernelBench tasks, demonstrating the effectiveness  
786 of deeply orchestrated, verification-driven agent pipelines for reliable kernel synthesis.

787 In contrast to large, highly structured multi-agent frameworks, Zhang et al. [42] propose **Cud-aForge**,  
788 a lightweight dual-agent system that separates kernel generation and evaluation into  
789 a Coder–Judge loop. The Coder generates CUDA kernel candidates based on task instructions  
790 and feedback from the Judge, while the Judge evaluates each candidate using correctness checks,  
791 runtime profiling, and hardware metrics (e.g., GPU specifications and Nsight Compute outputs)  
792 to identify bottlenecks and provide targeted optimization guidance. This iterative process allows  
793 the Coder to progressively refine kernels across multiple rounds, correcting errors and improving  
794 performance in a directed manner. By decoupling generation and evaluation, CudaForge achieves  
795 highest correctness rate and significant performance gains over baseline approaches on Kernel-  
796 bench [17] while maintaining strong practical performance. These results highlight that even a  
797 minimalist agentic decomposition, when combined with iterative, hardware-aware feedback, can  
798 deliver meaningful gains in real-world kernel optimization.

799 Building on lightweight, profiling agentic refinement such as CudaForge, Lei et al. [43] further  
800 propose **PRAGMA**, a multi-agent framework that tightly integrates fine-grained hardware profiling  
801 into the LLM optimization loop. Not only does PRAGMA rely on correctness or coarse runtime  
802 feedback, but grounds iterative kernel refinement in detailed, hardware-aware performance signals  
803 collected from both GPU and CPU backends. PRAGMA employs a Profiler Agent to gather low-  
804 level metrics from diverse profiling tools, including Nsight Compute and Linux perf. A dedicated  
805 Conductor Agent then interprets these metrics, performs bottleneck classification, and distills them  
806 into high-level optimization hints. Guided by this feedback, the Coder Agent iteratively refines  
807 kernel implementations, while the system explicitly preserves historically best-performing variants  
808 and their profiling traces, enabling context-aware reasoning over evolving performance bottlenecks.  
809 Experimental results on KernelBench [17] demonstrate that PRAGMA consistently outperforms  
810 prior LLM-based approaches, achieving averaged speedups of  $2.81 \times$  on CPU and  $2.30 \times$ – $4.50 \times$  on  
811 GPU, and up to  $10.95 \times$  over baseline LLM-generated kernels. These results highlight the effectiveness  
812 of reasoning based on detailed profiling feedback and explicit bottleneck interpretation.

813 Li et al. [50] propose **TritonForge**, a framework that centers on a LLM optimization pipeline for  
814 Triton kernels. TritonForge incorporates specialized agents for test generation, kernel optimization,  
815 and fault-aware remediation, forming a multi-stage workflow that supports automated benchmarking,  
816 error correction, and iterative refinement. Profiling and code generation are performed in a  
817 closed loop until performance converges or a predefined iteration budget is reached, enabling Tri-  
818 tonForge to progressively steer Triton kernels toward high-performance implementations without  
819 manual profiling expertise. Moreover, TritonForge also integrates NVIDIA Nsight Compute into  
820 the optimization loop to collect low-level hardware metrics, such as memory throughput, warp  
821 occupancy, and instruction stalls, and translates these profiling signals into structured feedback  
822 for the LLM. Based on this feedback, the model generates targeted code modifications, including  
823 changes to tiling strategies, memory layouts, and the insertion of auto-tuning directives. While this  
824 profiling-guided loop enables TritonForge to progressively steer kernels toward higher performance  
825 without manual profiling expertise, its iterative search exhibits limited exploration efficiency: the  
826 LLM often revisits semantically similar but performance-neutral variants and tends to converge  
827 prematurely to shallow performance plateaus, reflecting the lack of gradient-like guidance in  
828 profiling feedback and motivating the need for stronger diversity control, adaptive stopping, and  
829 memory-augmented search in future designs.

830 Furthermore, Sereda et al. [47] introduce **KForge**, which is a platform-agnostic agentic framework.  
831 KForge is designed to operate across diverse accelerator backends. It combines a generation agent  
832 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, Nagaitev et al. [48] 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** [51] 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 [35] attempts to extend its ability to autonomously apply a diverse set of optimization patterns across different kernels, KForge [47] 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.

### 4.3 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 -03-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

883 training cost, as reward signals must be obtained through repeated physical execution on GPUs.  
 884 Moreover, the lack of accurate analytical performance models for SASS-level instructions, limits  
 885 scalability and cross-domain generalization. Consequently, applying CuAsmRL to kernels from  
 886 new domains still requires domain-specific retraining and manual verification.

887 **KernelLLM** [25] curated the KernelBook dataset and employs SFT for end-to-end Triton kernel  
 888 generation. KernelLLM fine-tunes Llama-3.1-8B-Instruct on approximately 25,000 paired examples  
 889 of PyTorch modules and their corresponding Triton kernel implementations, augmented with  
 890 synthetically compiled samples generated via `torch.compile()` and curated code from TheStack  
 891 [56]. The resulting dataset, KernelBook, provides structured supervision that explicitly aligns  
 892 high-level PyTorch semantics with low-level Triton implementations. Trained using standard  
 893 instruction-based SFT, KernelLLM translates PyTorch programs into Triton kernel candidates,  
 894 which are validated through unit tests and pass@k sampling on KernelBench-Triton. Despite its  
 895 relatively modest parameter scale, KernelLLM achieves competitive performance with significantly  
 896 larger frontier models, highlighting the effectiveness of curated supervision in imparting GPU  
 897 programming patterns. However, as an imitation-based SFT approach, KernelLLM primarily inherits  
 898 the optimization strategies present in the training corpus, limiting its ability to extrapolate beyond  
 899 observed optimization patterns.

900 **Kevin (Kernel Devin)** [30] pioneers multi-turn reinforcement learning for CUDA kernel genera-  
 901 tion. For each task, Kevin samples multiple parallel trajectories, where kernels are iteratively refined  
 902 over several turns. Each refinement turn consists of a chain-of-thought (CoT) reasoning step and a  
 903 kernel generation step, where the CoT verbalizes intermediate optimization decisions, while the  
 904 kernel generation step concretely implements these decisions into an updated CUDA kernel, and  
 905 is treated as an individual training sample with execution-grounded rewards. To prevent context  
 906 explosion, long CoTs are discarded while compact summaries of optimization actions, together  
 907 with previously generated kernels and evaluation feedback, are retained to condition subsequent  
 908 refinement turns. Evaluated on KernelBench [17], Kevin improves kernel correctness from 56%  
 909 to 82% and increases mean speedup from 0.53× to 1.10× over the PyTorch Eager baseline. These  
 910 results demonstrate that reinforcement learning can effectively train models to reason and optimize  
 911 over a sequence of structured refinement steps. However, its evaluation is primarily conducted on  
 912 NVIDIA A100 GPUs, leaving generalization to diverse hardware architectures as an open question.

913 Following Kevin, **CUDA-L1** introduced by Li et al. [31] includes three stages: Supervised Fine-  
 914 Tuning with Data Augmentation, Self-Supervised Learning, and Contrastive Reinforcement Learn-  
 915 ing. The approach augments the training dataset with CUDA code variants generated by LLMs  
 916 and fine-tunes the base model on executable and correct implementations to establish founda-  
 917 tional CUDA knowledge. The model then iteratively generates CUDA kernels, validates their  
 918 correctness and executability, and trains on successfully validated examples, enabling autonomous  
 919 improvement without human supervision. Additionally, contrastive learning is employed with  
 920 execution-time rewards, training the model to distinguish between faster and slower CUDA im-  
 921 plementations, ultimately optimizing for superior performance. However, the CUDA-L1 approach  
 922 relies on iterative generation, validation, and training cycles, which makes the whole process  
 923 relatively time-consuming.

924 In the triton domain, Li et al. [33] introduce **AutoTriton** that represents the first dedicated RL-  
 925 trained model for Triton kernel synthesis, combining SFT with Group Relative Policy Optimization  
 926 (GRPO) reinforcement learning [57] under hybrid rewards based on rule and execution. Built on an  
 927 8B parameter architecture, AutoTriton first undergoes supervised fine-tuning on curated Triton  
 928 examples, then is further optimized using the GRPO algorithm with a hybrid reward function  
 929 that combines rule-based and execution-based feedback. AutoTriton demonstrates performance  
 930 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. [36] 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. [39] 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. [40] 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 [25] and AutoTriton[33], 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** [44] 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. [46] 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.

#### 4.4 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.

#### 5 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

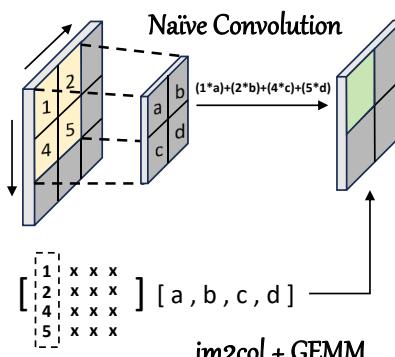
1030 advances in general-purpose code models have led to substantial improvements in overall coding  
 1031 capabilities, these models are not specifically adapted for kernel-level optimization, leaving consid-  
 1032 erable room to adapt and extend such advances to kernel-specific generation and optimization. In  
 1033 the following, we accordingly organize the roadmap into two subsections.

## 1034 5.1 Integrating Human Expertise

1035 This subsection provides a systematic summary of prior human expertise that can inform perfor-  
 1036 mance optimization in LLMs. In particular, we emphasize key perspectives of human expertise,  
 1037 including mathematical equivalence transformations, data locality optimization, hardware instruc-  
 1038 tion mapping, dynamic-to-static transformations, and precision–resource trade-offs. We illustrate  
 1039 these perspectives with a  $2 \times 2$  convolution on a  $3 \times 3$  input along the PyTorch-to-GPU execution  
 1040 pathway.

1041  
 1042 *5.1.1 Mathematical Equivalence Transformations.* Mathematical equivalence transformations refor-  
 1043 mulate computational problems into mathematically equivalent forms that can be implemented  
 1044 more efficiently.

1045 We illustrate this principle using the im2col transformation as an example. As showed in Figure 4,  
 1046 with a  $3 \times 3$  input and a  $2 \times 2$  kernel, im2col flattens each overlapping patch into a column and  
 1047 concatenates them into a matrix—converting convolution from scattered dot products into a single  
 1048 dense matrix multiplication (GEMM). This restructuring enables the use of optimized GEMM  
 1049 libraries and hardware accelerators.



1051  
 1052  
 1053 Fig. 4. Schematic comparison of native convolution versus im2col+GEMM convolution. Up: Native convolution  
 1054 performs a sliding-window dot product: the kernel  $[a, b, c, d]$  convolves with the input patch  $[1, 2, 4, 5]$  to  
 1055 produce one output element  $(1 \cdot a) + (2 \cdot b) + (4 \cdot c) + (5 \cdot d)$ . Down: im2col+GEMM first flattens all sliding  
 1056 windows into columns of a matrix (only the first column  $[1, 2, 4, 5]^T$  is shown), then multiplies the flattened  
 1057 kernel  $[a, b, c, d]$  with this matrix in a single GEMM operation, producing all output elements simultaneously.

1058  
 1059  
 1060  
 1061  
 1062  
 1063  
 1064  
 1065  
 1066  
 1067  
 1068  
 1069  
 1070  
 1071  
 1072  
 1073  
 1074  
 1075  
 1076  
 1077  
 1078 *5.1.2 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.

1079 Specifically, coalesced memory access allows threads to load contiguous data efficiently. Tiled  
1080 shared memory and kernel weight reuse in registers, combined with cache-aware layouts and bank-  
1081 conflict avoidance, can also provide significant benefits [58]. Specifically, data locality optimization  
1082 covers the following dimensions.

1083 **Coalesced Memory Access.** Coalescing ensures that consecutive threads within a warp access  
1084 consecutive memory locations, enabling a single wide memory transaction (e.g., 128 bytes) to  
1085 serve multiple threads efficiently. For regular access patterns, this can be achieved through proper  
1086 data layout, thread organization, on-chip memory utilization, and techniques such as reorganizing  
1087 threads [59, 60], selecting optimal thread block sizes [9], transforming data layouts (e.g., array of  
1088 structs to struct of arrays), and tiling [61–63]. For irregular access patterns, such as those in sparse  
1089 matrices, specialized data formats are required to maintain coalesced memory access [64].

1090 **Shared Memory Tiling.** Tiling (or spatial blocking) partitions data into blocks that fit within a  
1091 streaming multiprocessor’s shared memory, enabling high-bandwidth data reuse across multiple  
1092 computations. This technique is particularly effective for operations with regular access patterns  
1093 such as matrix multiplication and convolution. Shared memory tiling exploits both temporal  
1094 locality (reusing data across multiple operations) [65–67] and spatial locality (accessing nearby  
1095 data) [68–70].

1096 **Kernel Fusion.** Kernel fusion merges multiple consecutive kernels into a single kernel, elim-  
1097 inating intermediate global memory writes and reads. This optimization reduces both memory  
1098 traffic and kernel launch overhead. Key benefits include improved data reuse and enhanced cache  
1099 utilization [71]. However, fusion may increase register and shared memory pressure, requiring  
1100 careful trade-off analysis [72].

1101 **Register Blocking.** Registers provide the fastest memory tier, with zero access latency. Register  
1102 blocking (or temporal blocking) stores frequently reused values (e.g., kernel weights in convolution)  
1103 in registers throughout computation; therefore this technique is especially effective for algorithms  
1104 with high temporal locality [73, 74].

1105 **Prefetching.** To reduce long memory latencies, data prefetching loads data for future computa-  
1106 tion steps before they are needed, overlapping memory transfers with computation. This technique  
1107 is commonly applied in dense linear algebra kernels (e.g., matrix multiplication [75]) and stencil  
1108 operations [66], often in combination with tiling and double buffering [76].

1109  
1110  
1111 **5.1.3 Hardware Instruction Optimization.** Hardware instruction optimization focuses on mapping  
1112 computation to efficient GPU instructions and scheduling them to maximize execution-unit util-  
1113 ization. At this level, the algorithmic structure and data layout are largely fixed, and performance  
1114 improvements are achieved by exploiting instruction-level parallelism and hardware-specific ex-  
1115 ecution characteristics. Specifically, optimization must address two fundamental tensions: ① the  
1116 severe mismatch between peak arithmetic throughput and long memory and instruction latencies,  
1117 and ② the contention for limited on-chip resources (e.g., registers and shared memory) induced  
1118 by massive thread-level parallelism. Effective instruction-level optimizations therefore require  
1119 careful coordination of instruction selection, scheduling, and resource allocation to fully exploit  
1120 the underlying GPU microarchitecture.

1121 Considerable human effort has historically gone into addressing the two fundamental perfor-  
1122 mance tensions in GPU kernels. To mitigate the mismatch between high arithmetic through-  
1123 put and long memory and instruction latencies, expert developers maximized parallelism at multiple  
1124 levels. At the instruction level, they applied loop unrolling and instruction scheduling to increase  
1125 instruction-level parallelism (ILP) and keep deep CUDA core pipelines occupied. Vectorization (e.g.,  
1126 using float4) further enhanced throughput by enabling SIMD execution and improving memory

1128 coalescing [77, 78]. At the thread-group level, warp-centric programming and warp shuffle operations  
 1129 facilitated efficient data exchange and reduction without shared-memory synchronization,  
 1130 sustaining warp activity and improving latency hiding [79, 80].

1131 Similarly, managing contention for limited on-chip resources required careful tuning of work  
 1132 granularity and execution mapping. Thread coarsening illustrated a key trade-off: assigning more  
 1133 computation per thread improved ILP and register-level data reuse but increased register pressure,  
 1134 potentially reducing occupancy and limiting latency hiding [81]. Likewise, offloading computation to  
 1135 specialized execution units such as Tensor Cores achieved orders-of-magnitude higher throughput  
 1136 for mixed-precision GEMM [82], but imposed strict constraints on data layout, problem size, and  
 1137 kernel design, influencing overall resource allocation and scheduling decisions.

1138 Previous work has shown that reducing numerical precision can improve computational efficiency  
 1139 and resource utilization, offering strategies that may also benefit large-scale model training and  
 1140 deployment. To balance computation, memory usage, and accuracy, practitioners selectively reduced  
 1141 precision, using FP16 and INT8 operations to leverage specialized hardware while maintaining  
 1142 acceptable model accuracy.

1143 During training, techniques such as Automatic Mixed Precision (AMP) dynamically combine  
 1144 16-bit and 32-bit operations, achieving significant speedups and memory savings without compro-  
 1145 mising accuracy [83]. For deployment, aggressive post-training quantization maps weights and  
 1146 activations to low-bit representations (e.g., INT4), reducing model size and enabling execution  
 1147 on resource-constrained devices, while quantization-aware training mitigates accuracy loss by  
 1148 embedding rounding and clipping directly into the forward pass [84]. Beyond precision alone,  
 1149 co-designing data layouts with reduced-precision arithmetic—such as transforming from Array-of-  
 1150 Structs to Struct-of-Arrays—has been shown to unlock additional speedups on GPUs by improving  
 1151 memory access patterns and cache utilization [85].

1152 Current LLMs fall short of fully understanding human domain expertise. Integrating such  
 1153 expertise in an agentic way could substantially improve LLM-driven kernel generation.

## 1154 5.2 Adapting Technical Advances in General-Purpose Code LLMs

1155 In this section, we distill recent advances in the general code domain into a set of high-level  
 1156 principles (P1–P3) that may guide the advancement of DL kernel generation.

- 1157 • **P1: Execution Semantic Integration.** Most LLM-based code generation approaches predomi-  
 1158 nantly operate on textual program representations, implicitly treating source code as a sufficient  
 1159 proxy for execution semantics relevant to optimization. However, in general programming tasks,  
 1160 it has been shown that textual representations alone are often insufficient to ensure functional  
 1161 correctness.

1162 To address this limitation, some studies have explored ways to embed richer code semantics  
 1163 into model training, enabling LLMs to better capture execution-level behaviors and performance-  
 1164 relevant properties [86, 87]. This suggests that, for kernel generation, explicitly incorporating  
 1165 semantic information such as execution-level behaviors from kernel code into LLM-based methods  
 1166 could substantially enhance their ability to generate functionally correct kernels.

- 1167 • **P2: Performance-Aware Semantic Integration.** Performance-oriented code generation, such  
 1168 as kernel optimization, differs from standard code generation in that it requires balancing runtime  
 1169 efficiency and functional correctness, which could be framed as a multi-objective optimization  
 1170 problem. Recent advances [88–90] have been proposed to address multi-objective optimization in  
 1171 the general-purpose code domain. These developments also suggest a path forward for LLM-based  
 1172 kernel generation.

- 1173 • **P3: Hardware-Aware Cost Modeling.** Kernel optimization is inherently hardware-aware: per-  
 1174 formance depends critically on factors such as memory access patterns, cache behavior, parallelism,  
 1175 1176

1177 and synchronization. Current LLMs, however, lack native awareness of these hardware constraints,  
 1178 limiting their ability to reliably generate high-performance kernels. Recent advances [91] have  
 1179 focused on predicting numeric outcomes of code executions, which has the potential to be applied  
 1180 in hardware-aware cost modeling. This could be further leveraged to improve hardware-aware  
 1181 kernel generation in LLMs.

1182

## 1183 6 Conclusion

1184 This survey provides an overview of current benchmarks and techniques for LLM-driven kernel  
 1185 generation and optimization. Despite notable progress, systematically improving the performance  
 1186 of existing methods remains challenging. We summarize insights from the pre-LLM era and the  
 1187 broader code generation domain that may inform future advances in kernel optimization. We hope  
 1188 this survey serves as a timely reference and motivates further research on DL kernel generation  
 1189 and optimization.

1190

## 1191 References

- [1] Jaime Sevilla, Lennart Heim, Anson Ho, Tamay Besiroglu, Marius Hobbahn, and Pablo Villalobos. 2022. Compute trends across three eras of machine learning. In *2022 international joint conference on neural networks (IJCNN)*. IEEE, 1–8.
- [2] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*. 1–12.
- [3] Sarunas Kalade and Graham Schelle. 2025. NPUEval: Optimizing NPU Kernels with LLMs and Open Source Compilers. *arXiv preprint arXiv:2507.14403* (2025).
- [4] Zixu Hao, Jianyu Wei, Tuowei Wang, Minxing Huang, Huiqiang Jiang, Shiqi Jiang, Ting Cao, and Ju Ren. 2025. Scaling LLM Test-Time Compute with Mobile NPU on Smartphones. *arXiv preprint arXiv:2509.23324* (2025).
- [5] Minh-Khoi Nguyen-Nhat, Hoang Duy Nguyen Do, Huyen Thao Le, and Thanh Tuan Dao. 2024. 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, 1–8.
- [6] Gregory Bolet, Giorgis Georgakoudis, Harshitha Menon, Konstantinos Parasyris, Nirajan Hasabnis, Hayden Estes, Kirk Cameron, and Gal Oren. 2025. Can large language models predict parallel code performance?. In *Proceedings of the 34th International Symposium on High-Performance Parallel and Distributed Computing*. 1–6.
- [7] Christoforos Kachris. 2025. A survey on hardware accelerators for large language models. *Applied Sciences* 15, 2 (2025), 586.
- [8] Sunpyo Hong and Hyesoon Kim. 2009. 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*. 152–163.
- [9] Arno Leist, Daniel P Playne, and Kenneth A Hawick. 2009. Exploiting graphical processing units for data-parallel scientific applications. *Concurrency and Computation: Practice and Experience* 21, 18 (2009), 2400–2437.
- [10] Benjamin F Spector, Simran Arora, Aaryan Singh, Daniel Y Fu, and Christopher Ré. 2024. Thunderkittens: Simple, fast, and adorable ai kernels. *arXiv preprint arXiv:2410.20399* (2024).
- [11] Guoliang He and Eiko Yoneki. 2025. CuAsmRL: Optimizing GPU SASS Schedules via Deep Reinforcement Learning. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*. 493–506.
- [12] Arya Tschand, Muhammad Awad, Ryan Swann, Kesavan Ramakrishnan, Jeffrey Ma, Keith Lowery, Ganesh Dasika, and Vijay Janapa Reddi. 2025. Swizzleperf: Hardware-aware llms for gpu kernel performance optimization. *arXiv preprint arXiv:2508.20258* (2025).
- [13] Pieter Hijma, Stijn Heldens, Alessio Scocco, Ben Van Werkhoven, and Henri E Bal. 2023. Optimization techniques for GPU programming. *Comput. Surveys* 55, 11 (2023), 1–81.
- [14] Alfred Santa Molison, Marcia Moraes, Glauzia Melo, Fabio Santos, and Wesley KG Assuncao. 2025. Is llm-generated code more maintainable\& reliable than human-written code. *arXiv preprint arXiv:2508.00700* (2025).
- [15] Blesson Varghese, Nan Wang, David Bermbach, Cheol-Ho Hong, Eyal De Lara, Weisong Shi, and Christopher Stewart. 2021. A survey on edge performance benchmarking. *ACM Computing Surveys (CSUR)* 54, 3 (2021), 1–33.
- [16] Triton Development Community. 2026. Triton Documentation. <https://triton-lang.org/main/index.html>. Accessed: 2026-01.
- [17] Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. 2025. Kernelbench: Can llms write efficient gpu kernels? *arXiv preprint arXiv:2502.10517* (2025).

1225

- [18] Zhongzhen Wen, Yinghui Zhang, Zhong Li, Zhongxin Liu, Linna Xie, and Tian Zhang. 2025. MultiKernelBench: A Multi-Platform Benchmark for Kernel Generation. *arXiv e-prints* (2025), arXiv-2507.
- [19] Robert Tjarko Lange, Qi Sun, Aaditya Prasad, Maxence Falldor, Yujin Tang, and David Ha. 2025. Towards robust agentic cuda kernel benchmarking, verification, and optimization. *arXiv preprint arXiv:2509.14279* (2025).
- [20] Jianling Li, Shangzhan Li, Zhenye Gao, Qi Shi, Yuxuan Li, Zefan Wang, Jiacheng Huang, WangHaojie WangHaojie, Jianrong Wang, Xu Han, et al. 2025. Tritonbench: Benchmarking large language model capabilities for generating triton operators. In *Findings of the Association for Computational Linguistics: ACL 2025*. 23053–23066.
- [21] NVIDIA. 2025. compute-eval. [Online]. Available: <https://github.com/NVIDIA/compute-eval>. Accessed: Dec. 21, 2025.
- [22] Meta. 2025. BackendBench. [Online]. Available: <https://github.com/meta-pytorch/BackendBench>. Accessed: Dec. 21, 2025.
- [23] Terry Chen, Bing Xu, and Kirthi Devleker. 2025. 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/>. Accessed: Nov. 11, 2025.
- [24] Matyáš Brabec, Jiří Klepl, Michal Töpfer, and Martin Kruliš. 2025. Tutoring LLM into a Better CUDA Optimizer. In *European Conference on Parallel Processing*. Springer, 250–263.
- [25] Zacharias V. Fischers, Sahan Paliskara, Simon Guo, Alex Zhang, Joe Spisak, Chris Cummins, Hugh Leather, Gabriel Synnaeve, Joe Isaacson, Aram Markosyan, and Mark Saroufim. 2025. KernelLLM: Making Kernel Development More Accessible. [Online]. Available: <https://huggingface.co/facebook/KernelLLM>. Accessed: Nov. 11, 2025.
- [26] Wentao Chen, Jiace Zhu, Qi Fan, Yehan Ma, and An Zou. 2025. CUDA-LLM: LLMs Can Write Efficient CUDA Kernels. *arXiv preprint arXiv:2506.09092* (2025).
- [27] NVIDIA Corporation. 2025. Cuda code samples. [Online]. Available: <https://github.com/NVIDIA/cuda-samples>. Accessed: Dec. 18, 2025.
- [28] LeetGPU. 2025. Challenges. [Online]. Available: <https://leetgpu.com/challenges>. Accessed: Dec. 18, 2025.
- [29] Martin Andrews and Sam Witteveen. 2025. GPU Kernel Scientist: An LLM-Driven Framework for Iterative Kernel Optimization. *arXiv preprint arXiv:2506.20807* (2025).
- [30] Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. 2025. Kevin: Multi-turn rl for generating cuda kernels. *arXiv preprint arXiv:2507.11948* (2025).
- [31] Xiaoya Li, Xiaofei Sun, Albert Wang, Jiwei Li, and Chris Shum. 2025. Cuda-l1: Improving cuda optimization via contrastive reinforcement learning. *arXiv preprint arXiv:2507.14111* (2025).
- [32] Jianghui Wang, Vinay Joshi, Saptarshi Majumder, Xu Chao, Bin Ding, Ziqiong Liu, Pratik Prabhjanan Brahma, Dong Li, Zicheng Liu, and Emad Barsoum. 2025. Geak: Introducing triton kernel ai agent & evaluation benchmarks. *arXiv preprint arXiv:2507.23194* (2025).
- [33] Shangzhan Li, Zefan Wang, Ye He, Yuxuan Li, Qi Shi, Jianling Li, Yonggang Hu, Wanxiang Che, Xu Han, Zhiyuan Liu, et al. 2025. Autotriton: Automatic triton programming with reinforcement learning in llms. *arXiv preprint arXiv:2507.05687* (2025).
- [34] Shikhar Mishra and Ayush Nangia. 2025. 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>. Accessed: 2025-11-12.
- [35] Anjiang Wei, Tianran Sun, Yogesh Seenichamy, Hang Song, Anne Ouyang, Azalia Mirhoseini, Ke Wang, and Alex Aiken. 2025. Astra: A multi-agent system for gpu kernel performance optimization. *arXiv preprint arXiv:2509.07506* (2025).
- [36] Lingcheng Kong, Jiateng Wei, Hanzhang Shen, and Huan Wang. 2025. ConCuR: Conciseness Makes State-of-the-Art Kernel Generation. *arXiv preprint arXiv:2510.07356* (2025).
- [37] Ping Guo, Chenyu Zhu, Siyuan Chen, Fei Liu, Xi Lin, Zhichao Lu, and Qingfu Zhang. 2025. EvoEngineer: Mastering Automated CUDA Kernel Code Evolution with Large Language Models. *arXiv preprint arXiv:2510.03760* (2025).
- [38] Juncheng Dong, Yang Yang, Tao Liu, Yang Wang, Feng Qi, Vahid Tarokh, Kaushik Rangadurai, and Shuang Yang. 2025. STARK: Strategic Team of Agents for Refining Kernels. *arXiv preprint arXiv:2510.16996* (2025).
- [39] Daniel Nichols, Konstantinos Parasyris, Charles Jekel, Abhinav Bhatele, and Harshitha Menon. 2025. Integrating Performance Tools in Model Reasoning for GPU Kernel Optimization. *arXiv preprint arXiv:2510.17158* (2025).
- [40] Jiin Woo, Shaowei Zhu, Allen Nie, Zhen Jia, Yida Wang, and Youngsuk Park. 2025. TritonRL: Training LLMs to Think and Code Triton Without Cheating. *arXiv preprint arXiv:2510.17891* (2025).
- [41] Laura Wang and the PyTorch Team at Meta. 2025. KernelFalcon: Autonomous GPU Kernel Generation via Deep Agents. [Online]. Available: <https://pytorch.org/blog/kernelfalcon-autonomous-gpu-kernel-generation-via-deep-agents/>. Accessed: Nov. 11, 2025.
- [42] Zijian Zhang, Rong Wang, Shiyang Li, Yuebo Luo, Mingyi Hong, and Caiwen Ding. 2025. CudaForge: An Agent Framework with Hardware Feedback for CUDA Kernel Optimization. *arXiv preprint arXiv:2511.01884* (2025).
- [43] Kelun Lei, Hailong Yang, Huaitao Zhang, Xin You, Kaige Zhang, Zhongzhi Luan, Yi Liu, and Depei Qian. 2025. PRAGMA: A Profiling-Reasoned Multi-Agent Framework for Automatic Kernel Optimization. *arXiv preprint arXiv:2511.06345*

- 1275 (2025).
- 1276 [44] Anonymous. 2025. Mastering Sparse CUDA Generation through Pretrained Models and Deep Reinforcement Learning.  
1277 In *Submitted to The Fourteenth International Conference on Learning Representations*. <https://openreview.net/forum?id=VdLEaGPYWT> under review.
- 1278 [45] Dezhi Ran, Shuxiao Xie, Mingfang Ji, Ziyue Hua, Mengzhou Wu, Yuan Cao, Yuzhe Guo, Yu Hao, Linyi Li, Yitao Hu, et al.  
1279 2025. KernelBand: Boosting LLM-based Kernel Optimization with a Hierarchical and Hardware-aware Multi-armed  
1280 Bandit. *arXiv preprint arXiv:2511.18868* (2025).
- 1281 [46] Xinguo Zhu, Shaohui Peng, Jiaming Guo, Yunji Chen, Qi Guo, Yuanbo Wen, Hang Qin, Ruizhi Chen, Qirui Zhou, Ke  
1282 Gao, et al. 2025. QiMeng-Kernel: Macro-Thinking Micro-Coding Paradigm for LLM-Based High-Performance GPU  
1283 Kernel Generation. *arXiv preprint arXiv:2511.20100* (2025).
- 1284 [47] Taras Sereda, Tom St John, Burak Bartan, Natalie Serrino, Sachin Katti, and Zain Asgar. 2025. KForge: Program  
1285 Synthesis for Diverse AI Hardware Accelerators. *arXiv preprint arXiv:2511.13274* (2025).
- 1286 [48] Kirill Nagaitsev, Luka Grbcic, Samuel Williams, and Costin Iancu. 2025. Optimizing PyTorch Inference with LLM-Based  
1287 Multi-Agent Systems. *arXiv preprint arXiv:2511.16964* (2025).
- 1288 [49] METR. 2025. Measuring Automated Kernel Engineering. [Online]. Available: <https://metr.org/blog/2025-02-14-measuring-automated-kernel-engineering/>. Accessed: Dec. 18, 2025.
- 1289 [50] Haonan Li, Keyu Man, Partha Kanuparth, Hanning Chen, Wei Sun, Sreen Tallam, Chenguang Zhu, Kevin Zhu, and  
1290 Zhiyun Qian. 2025. TritonForge: Profiling-Guided Framework for Automated Triton Kernel Optimization. *arXiv  
1291 preprint arXiv:2512.09196* (2025).
- 1292 [51] MindSpore. 2025. Auto Kernel Generator (AKG). [Online]. Available: <https://atomgit.com/mindspore/akg>. Accessed:  
1293 Dec. 18, 2025.
- 1294 [52] OpenAI. 2024. GPT-4o System Card. *arXiv:2410.21276* [cs.CL] <https://arxiv.org/abs/2410.21276>
- 1295 [53] Deepseek-AI. 2025. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature* 645, 8081  
1296 (Sept. 2025), 633–638. doi:10.1038/s41586-025-09422-z
- 1297 [54] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste  
1298 Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume  
1299 Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. *arXiv:2302.13971* [cs.CL] <https://arxiv.org/abs/2302.13971>
- 1300 [55] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos  
1301 Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2024. Sglang: Efficient execution of structured language model programs.  
1302 *Advances in neural information processing systems* 37 (2024), 62557–62583.
- 1303 [56] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite,  
1304 Margaret Mitchell, Sean Hughes, Thomas Wolf, et al. 2022. The stack: 3 tb of permissively licensed source code. *arXiv  
1305 preprint arXiv:2211.15533* (2022).
- 1306 [57] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Huawei Zhang, Mingchuan Zhang, YK Li,  
1307 Yang Wu, et al. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv  
1308 preprint arXiv:2402.03300* (2024).
- 1309 [58] Pieter Hijma, Stijn Heldens, Alessio Scocco, Ben Van Werkhoven, and Henri E Bal. 2023. Optimization techniques for  
1310 GPU programming. *Comput. Surveys* 55, 11 (2023), 1–81.
- 1311 [59] Juan Gómez-Luna, José María González-Linares, José Ignacio Benavides, and Nicolás Guil. 2013. An optimized approach  
1312 to histogram computation on GPU. *Machine Vision and Applications* 24, 5 (2013), 899–908.
- 1313 [60] Yang Hu, Hang Liu, and H Howie Huang. 2018. Tricore: Parallel triangle counting on gpus. In *SC18: International  
1314 Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 171–182.
- 1315 [61] Rajib Nath, Stanimire Tomov, Tingxing "Tim" Dong, and Jack Dongarra. 2011. Optimizing symmetric dense matrix-  
1316 vector multiplication on GPUs. In *Proceedings of 2011 International Conference for High Performance Computing,  
Networking, Storage and Analysis*. 1–10.
- 1317 [62] Wai Teng Tang, Wen Jun Tan, Rajarshi Ray, Yi Wen Wong, Weiguang Chen, Shyh-hao Kuo, Rick Siow Mong Goh,  
1318 Stephen John Turner, and Weng-Fai Wong. 2013. Accelerating sparse matrix-vector multiplication on GPUs using  
1319 bit-representation-optimized schemes. In *Proceedings of the International Conference on High Performance Computing,  
Networking, Storage and Analysis*. 1–12.
- 1320 [63] Rajib Nath, Stanimire Tomov, and Jack Dongarra. 2010. An improved magma gemm for fermi graphics processing  
1321 units. *The International Journal of High Performance Computing Applications* 24, 4 (2010), 511–515.
- 1322 [64] Jianlong Zhong and Bingsheng He. 2013. Medusa: Simplified graph processing on GPUs. *IEEE Transactions on Parallel  
and Distributed Systems* 25, 6 (2013), 1543–1552.
- 1323 [65] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. 2016. Performance tuning and optimization  
1324 techniques of fixed and variable size batched Cholesky factorization on GPUs. *Procedia Computer Science* 80 (2016),  
1325 119–130.

- [66] Justin Holewinski, Louis-Noël Pouchet, and Ponnuswamy Sadayappan. 2012. High-performance code generation for stencil computations on GPU architectures. In *Proceedings of the 26th ACM international conference on Supercomputing*. 311–320.
- [67] Youngdong Do, Hyungmo Kim, Pyeongseok Oh, Daeyoung Park, and Jaejin Lee. 2019. SNU-NPB 2019: parallelizing and optimizing NPB in OpenCL and CUDA for modern GPUs. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 93–105.
- [68] Minquan Fang, Jianbin Fang, Weimin Zhang, Haifang Zhou, Jianxing Liao, and Yuangang Wang. 2018. Benchmarking the GPU memory at the warp level. *Parallel Comput.* 71 (2018), 23–41.
- [69] Filip Petrovič, David Střelák, Jana Hozzová, Jaroslav Ol'ha, Richard Trembecký, Siegfried Benkner, and Jiří Filipovič. 2020. A benchmark set of highly-efficient CUDA and OpenCL kernels and its dynamic autotuning with Kernel Tuning Toolkit. *Future Generation Computer Systems* 108 (2020), 161–177.
- [70] Jianyu Huang, Chenhan D Yu, and Robert A van de Geijn. 2020. Strassen's algorithm reloaded on GPUs. *ACM Transactions on Mathematical Software (TOMS)* 46, 1 (2020), 1–22.
- [71] Matthias Korch and Tim Werner. 2018. Accelerating explicit ODE methods on GPUs by kernel fusion. *Concurrency and Computation: Practice and Experience* 30, 18 (2018), e4470.
- [72] Jesús Carabaño, Jan Westerholm, and Tapani Sarjakoski. 2018. A compiler approach to map algebra: automatic parallelization, locality optimization, and GPU acceleration of raster spatial analysis. *GeoInformatica* 22, 2 (2018), 211–235.
- [73] Nhat-Phuong Tran, Myungho Lee, and Dong Hoon Choi. 2015. Memory-efficient parallelization of 3D lattice Boltzmann flow solver on a GPU. In *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*. IEEE, 315–324.
- [74] Weidong Qiu, Zheng Gong, Yidong Guo, Bozhong Liu, Xiaoming Tang, and Yuheng Yuan. 2016. GPU-Based High Performance Password Recovery Technique for Hash Functions. *J. Inf. Sci. Eng.* 32, 1 (2016), 97–112.
- [75] Pham Nguyen Quang Anh, Rui Fan, and Yonggang Wen. 2016. Balanced hashing and efficient gpu sparse general matrix-matrix multiplication. In *Proceedings of the 2016 International Conference on Supercomputing*. 1–12.
- [76] Kiran Matam, Siva Rama Krishna Bharadwaj Indarapu, and Kishore Kothapalli. 2012. Sparse matrix-matrix multiplication on modern architectures. In *2012 19th International Conference on High Performance Computing*. IEEE, 1–10.
- [77] Ahmed A Abdelrahman, Mohamed M Fouad, Hisham Dahshan, and Ahmed M Mousa. 2017. High performance CUDA AES implementation: A quantitative performance analysis approach. In *2017 Computing conference*. IEEE, 1077–1085.
- [78] Gert-Jan van den Braak, Bart Mesman, and Henk Corporaal. 2010. Compile-time GPU memory access optimizations. In *2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*. IEEE, 200–207.
- [79] Michael Bauer, Henry Cook, and Brucek Khailany. 2011. CudaDMA: optimizing GPU memory bandwidth via warp specialization. In *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*. 1–11.
- [80] Takumi Honda, Yasuaki Ito, and Koji Nakano. 2015. A warp-synchronous implementation for multiple-length multiplication on the GPU. In *2015 Third International Symposium on Computing and Networking (CANDAR)*. IEEE, 96–102.
- [81] Joseph D Garvey and Tarek S Abdelrahman. 2018. A strategy for automatic performance tuning of stencil computations on GPUs. *Scientific Programming* 2018, 1 (2018), 6093054.
- [82] Da Yan, Wei Wang, and Xiaowen Chu. 2020. Demystifying tensor cores to optimize half-precision matrix multiply. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 634–643.
- [83] Md Mehrab Hossain Opi, Sumaiya Khan, and Moshammad Farzana Rahman. 2025. Accelerating Bangla NLP Tasks with Automatic Mixed Precision: Resource-Efficient Training Preserving Model Efficacy. *arXiv preprint arXiv:2512.00829* (2025).
- [84] Enkhbold Nyamsuren. 2024. Evaluating Quantized Large Language Models for Code Generation on Low-Resource Language Benchmarks. *arXiv preprint arXiv:2410.14766* (2024).
- [85] Paweł K Radtke and Tobias Weinzierl. 2025. Compiler-supported reduced precision and AoS-SoA transformations for heterogeneous hardware. *arXiv preprint arXiv:2512.05516* (2025).
- [86] Junlong Li, Daya Guo, Dejian Yang, Runxin Xu, Yu Wu, and Junxian He. 2025. Codei/o: Condensing reasoning patterns via code input-output prediction. *arXiv preprint arXiv:2502.07316* (2025).
- [87] Yangruibo Ding, Jinjun Peng, Marcus Min, Gail Kaiser, Junfeng Yang, and Baishakhi Ray. 2024. Semcoder: Training code language models with comprehensive semantics reasoning. *Advances in Neural Information Processing Systems* 37 (2024), 60275–60308.
- [88] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515* (2024).
- [89] Mingzhe Du, Luu Anh Tuan, Yue Liu, Yuhao Qing, Dong Huang, Xinyi He, Qian Liu, Zejun Ma, and See-kiong Ng. 2025. Afterburner: Reinforcement learning facilitates self-improving code efficiency optimization. *arXiv preprint arXiv:2501.00001* (2025).

- 1373                  *arXiv:2505.23387* (2025).
- 1374 [90] Jian Yang, Xianglong Liu, Weifeng Lv, Ken Deng, Shawn Guo, Lin Jing, Yizhi Li, Shark Liu, Xianzhen Luo, Yuyu Luo,  
1375                  et al. 2025. From Code Foundation Models to Agents and Applications: A Comprehensive Survey and Practical Guide  
1376                  to Code Intelligence. *arXiv preprint arXiv:2511.18538* (2025).
- 1377 [91] Yash Akhauri, Xingyou Song, Arissa Wongpanich, Bryan Lewandowski, and Mohamed S Abdelfattah. 2025. Regression  
1378                  language models for code. *arXiv preprint arXiv:2509.26476* (2025).
- 1379
- 1380
- 1381
- 1382
- 1383
- 1384
- 1385
- 1386
- 1387
- 1388
- 1389
- 1390
- 1391
- 1392
- 1393
- 1394
- 1395
- 1396
- 1397
- 1398
- 1399
- 1400
- 1401
- 1402
- 1403
- 1404
- 1405
- 1406
- 1407
- 1408
- 1409
- 1410
- 1411
- 1412
- 1413
- 1414
- 1415
- 1416
- 1417
- 1418
- 1419
- 1420
- 1421