KForge: LLM-Driven Cross-Platform Kernel Generation for AI Accelerators

Taras Sereda Burak Bartan Ankita Nayak Tom St. John
Natalie Serrino Zain Asgar
Gimlet Labs Inc., San Francisco, California, USA
{taras, bbartan, ankitanayak, tstjohn, nserrino, zasgar}@gimletlabs.ai

***Abstract*—Production inference increasingly targets a heterogeneous mix of accelerators. Agentic pipelines interleave reasoning, tool calls, and multi-agent coordination, each with distinct compute and memory profiles. For optimal efficiency, each stage should run on the accelerator best suited to it. This creates a systems challenge: each pipeline now requires high-performance kernels across a growing set of hardware backends and programming models. Writing these kernels by hand is time-consuming, demands deep low-level expertise, and does not scale as kernel complexity grows. Recently, Large Language Models (LLMs) have been leveraged for automatic kernel generation, but challenges in low-level code generation and cross-backend generalization persist. We present KForge, a cross-platform framework built around an iterative refinement loop driven by two collaborating LLM-based agents: a generation agent that produces and progressively refines kernels using compilation and correctness feedback, and a performance-analysis agent that interprets profiling data, from programmatic APIs to GUI-based tools, and emits recommendations that steer the next round of synthesis. The loop alternates between functional passes, which drive a candidate to correctness, and optimization passes, which close the performance gap to hand-tuned baselines. We evaluate KForge on two backends with very different baseline reference availability. On NVIDIA B200, KForge achieves a 2.12% improvement in end-to-end throughput compared to TensorRT-LLM on the gpt-oss-20b inference speed benchmark. On Intel Arc B580, KForge generates Triton kernels achieving a 5.13× geometric mean speedup over the faster of PyTorch eager and `torch.compile` on 37 GEMM + tail-ops workloads from KernelBench Level 2, primarily via operator fusion and mixed-precision execution.**

I. Introduction

The rise in agentic AI applications is reshaping how inference workloads are deployed. A single pipeline now interleaves LLM calls with reasoning, data retrieval, tool invocations, and multi-agent coordination, each with different compute, memory, and synchronization profiles. No single accelerator is the right fit for all of these stages at once. Production deployments hence need to assign each stage to its best-suited accelerator for optimal performance. This heterogeneity gives rise to a corresponding systems design challenge [3].

Realizing this mapping across heterogeneous platforms requires optimized implementations of the same core operators across many different devices and programming models. Writing high-performance compute kernels requires mastering parallel programming languages such as CUDA [14], Metal [2], Triton [19], SYCL [9], or CuTe DSL [16]. Porting kernels across accelerators is rarely a syntactic translation, as each platform exposes different compute capabilities, memory hierarchies, bandwidth limits, and communication costs, and as a result, the best implementation often differs across platforms.

Modern compilers and inference runtimes have made substantial progress in automating performance optimization. Systems such as `torch.compile` [1] and TensorRT-LLM [15] improve neural network execution through graph optimizations and automatic kernel fusion. Nevertheless, building high-performance kernels requires combining clever algorithmic techniques with careful hardware utilization, as demonstrated by works such as FlashAttention [5], [6], where integrating online softmax [12] with tiled attention computation and hardware-specific instructions enables superior performance. Together, these optimizations reduce memory traffic, improve arithmetic intensity, and avoid scheduling overheads that general-purpose compiler passes may not eliminate. End-to-end model inference complicates the problem further. A kernel that is faster in isolation may not improve full-model runtime if it prevents graph-level optimizations, introduces layout or precision conversions, or shifts bottlenecks elsewhere in the pipeline. Conversely, a kernel that appears modest in standalone benchmarks may unlock larger gains when composed with favorable surrounding operations. Kernel optimization must be evaluated both locally, at the level of individual kernels, and globally, in the context of full-model execution.

Recent progress in Large Language Models (LLMs) has made code generation increasingly plausible. However, extending these capabilities to kernel generation remains a challenge. Low-level performance code is generally brittle: small deltas in memory layout or numerical precision can break correctness or eliminate performance gains. Furthermore, training data for LLMs is skewed toward CUDA, leaving emerging platforms underrepresented.

In this work, we present KForge, a cross-platform framework for LLM-driven kernel generation and optimization, as described in Figure 1. KForge mirrors the real-world workflow of a kernel engineer through an iterative refinement loop. The system alternates between functional passes, which drive a candidate towards correctness, and optimization passes, which improve runtime once a correct implementation is reached. During functional passes, the generation agent proposes candidate kernels and uses compilation and correctness feedback to converge on a valid implementation. During optimization passes, the performance-analysis agent interprets profiling data,

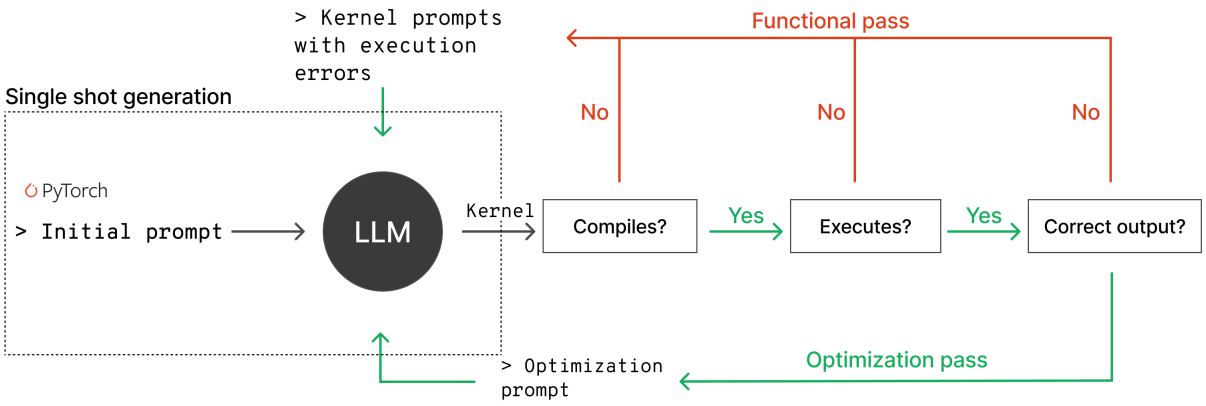

Fig. 1. Iterative program synthesis and optimization loop using LLMs. The workflow consists of two main phases: (1) a functional pass that iteratively refines synthesized programs until the code compiles, executes without errors, and produces correct output, and (2) an optimization pass that provides performance feedback to the LLM for iterative performance improvement.

including both programmatic metrics and GUI-based profiler outputs, and produces concrete code-level recommendations. These recommendations target hardware utilization metrics such as memory bandwidth utilization, warp occupancy, and arithmetic intensity, and feed into the next round of synthesis.

KForge is architected to generate kernels across diverse hardware backends and programming models. We evaluate the same agentic architecture across various backends and programming models, enabling examination of when kernel generation agents can transfer algorithmic ideas across platforms, and when they need backend-specific guidance to exploit hardware-specific capabilities. We evaluate KForge through two case studies chosen to probe very different baseline availability: NVIDIA B200, where KForge competes against years of vendor-tuned reference kernels in TensorRT-LLM, and Intel Arc B580, where comparable hand-tuned references do not exist, and the system instead serves as a bring-up tool for emerging hardware. In summary, the paper makes the following **key contributions**:

- We introduce KForge, a multi-stage autonomous program synthesis framework in which the generation and performance-analysis agents collaborate to produce correct and optimized kernels using compilation, correctness, and profiling feedback.
- We describe a system architecture that supports four accelerator vendors (NVIDIA, AMD, Apple, Intel) and six programming models (CUDA, Triton, CuTe DSL, HIP, SYCL, Metal) through a uniform program synthesis interface.
- We present case studies on two representative backends: end-to-end speedup over TensorRT-LLM on NVIDIA B200, and Triton kernel generation on Intel Arc B580 where comparable hand-tuned references are unavailable.

II. Related Work

LLM-driven kernel generation. A growing body of work uses LLMs to generate and optimize GPU kernels, predominantly targeting the NVIDIA ecosystem where training data on kernel development is abundant. Sakana AI's CUDA Engineer [10] used evolutionary search for automated CUDA kernel discovery. Follow-up analysis found that the system exploited evaluation framework vulnerabilities, illustrating the difficulty of robust evaluation in this space. CUDA-LLM [4] develops a Feature Search and Reinforcement (FSR) framework that combines compilation, correctness, and profiling feedback to refine CUDA kernels. KernelBlaster [7] augments a GPU coding agent with a persistent CUDA knowledge base accumulated across tasks. Autocomp [8] extends LLM-driven optimization beyond NVIDIA, targeting tensor accelerators more broadly through planned hardware optimizations and hardware feedback. However, most works tend to focus on per-kernel speedups rather than measuring impact on full-model execution.

Kernel-level benchmarks. Several benchmarks have been developed to evaluate LLM-generated kernels. KernelBench [18] introduced a benchmark framework with 250 PyTorch workloads to evaluate LLMs' ability to generate efficient GPU kernels. The benchmark uses a $fast_p$ metric measuring both correctness and speedup over baseline implementations. In contrast, NVIDIA's SOL-ExecBench [11] focuses on evaluating the theoretical lower limit using an analytical pipeline which incorporates the workload's FLOP count and byte count coupled with the target hardware's peak bandwidth/FLOP capabilities. While these benchmarks have advanced per-kernel evaluation, they leave open the question of how LLM-generated kernels affect end-to-end model performance once integrated into production runtimes such as TensorRT-LLM, where graph fusion, autotuning, and surrounding kernel selection interact with the kernel under study.

KForge generates kernels across four AI accelerators and six programming models through a uniform interface, and evaluates the impact of generated kernels on full-model execution against integrated production runtimes such as TensorRT-LLM.

III. KForge: Autonomous Program Synthesis

KForge is a multi-stage autonomous program synthesis framework, illustrated in Figure 1. It supports iterative refinement, single-shot program synthesis, as well as repetitive sampling, where the mode of operation is directed by prompt construction. It is designed as a cross-platform framework and currently supports a diverse set of hardware targets, spanning four vendors and six programming models with varying levels of abstraction, visualized in Figure 2. For NVIDIA GPUs, we generate kernels in CUDA, Triton, and CuTe DSL, covering both low-level hand-tuned and higher-level abstractions. For AMD GPUs, we support HIP and Triton. For Intel Arc GPUs, we target SYCL and Triton. For Apple silicon, we generate Metal kernels. KForge is also model-agnostic; the underlying LLM is selected through a model registry, allowing users to plug in any model without changes to the synthesis pipeline.

Within the scope of this work, we focus on the following three strategies that we employ for program synthesis. These strategies are complementary to one another, allowing one to build dynamic configurations based on available sources of supervision and computational resource budgets.

- **Iterative refinement.** It allows the model to make error corrections from the previous run or optimize the performance of the correctly generated kernel, taking into account the program synthesized in the previous iteration. Specifically, for each iteration $i \in \{1, \ldots, N-1\}$ we add evaluation results from iteration $i-1$ to the model's prompt, with a corresponding instruction to fix the error or improve program performance.
- **Cross-platform translation.** When a functional implementation already exists for one accelerator, it can be provided to the model as a reference, enabling cross-platform translation. For example, when generating Metal kernels, the prompt may include the corresponding CUDA implementation. This is particularly useful when the target backend lacks training data coverage.
- **Profiling feedback.** Profiling data is crucial for pinpointing bottlenecks, providing comprehensive information on hardware resource usage for a specific computational workload. Timeline views assist in identifying scheduling gaps, while detailed statistics at the level of individual accelerator API calls highlight parts of the computational graph that do not fully utilize hardware resources.

A. Program Synthesis Agent

We follow a task definition similar to that in [18]. Specifically, we treat the LLM as a function $F : (p) \mapsto k$ that receives a text prompt $p \in \mathcal{T}$ as input and returns the generated code $k \in \mathcal{T}$. The generated code is expected to contain a kernel program, a kernel scheduling code, a JIT-library compilation code, and a PyTorch model `class NewModel(nn.Module)` with a `def forward(self, *inputs)` method that implements the module's forward pass. We use the Jinja2 template engine to parameterize the prompts. The default prompt p contains a high-level task description, a one-shot example (a PyTorch implementation paired with its corresponding target-accelerator implementation), an input problem in PyTorch, and a task description in natural language.

B. Performance Analysis Agent

We introduce a specialized agent for performance analysis rather than handling generation and optimization in a single agent for two reasons. First, profiling data is extensive but optimization signals are sparse; previous research [13] shows that LLM performance on relevant information retrieval drops to 50% for 32K token inputs versus <1K tokens. Second, separating the two roles enables a modular architecture in which different models can be assigned to each agent based on their respective strengths.

The Performance Analysis Agent processes profiling inputs, such as raw metrics from NVIDIA Nsight Systems or visual outputs from Xcode Instruments, and generates optimization recommendations for subsequent program synthesis iterations. This platform-agnostic approach handles arbitrary textual or visual profiling data across different hardware accelerators.

Formally, the agent is defined as $G : (o, k, \{v^0, ..., v^n\}) \mapsto r$, where $o \in \mathcal{T}$ is the text performance optimization prompt; $k \in \mathcal{T}$ is the synthesized program; $v^i \in \mathbb{R}^{H \times W \times C} \cup \mathcal{T}, i \in \{0, ..., n\}$ represents profiling information as screenshots when $v^i \in \mathbb{R}^{H \times W \times C}$ or text-based profiler output when $v^i \in \mathcal{T}$; and $r \in \mathcal{T}$ is the performance recommendation. The agent is prompted to generate a single recommendation for maximum performance improvement. The recommendation r_t feeds into the next synthesis iteration, establishing a feedback loop: $F : (p, k_{t-1}, r_{t-1}) \mapsto k_t$.

C. Program Verification

The proposed execution flow defines a closed feedback loop, with information that is either helpful in recovering from failures or allows the model to optimize a functionally correct implementation to achieve speedup. After every generation-evaluation iteration, we save detailed logs for each workload. We focus on five possible execution states:

- *generation failure* — typical reasons: network error, model output does not contain workload's code.
- *compilation failure* — the generated result contains workload's code, but fails to compile.
- *runtime error* — the workload's code compiles but fails at runtime, typically caused by segmentation faults or program abort.
- *numerical or shape mismatch* — call to `NewModel.forward` returns tensors, but they mismatch in tensor shapes or expected values, or both.
- *correct* — call to `NewModel.forward` returns tensors that match expected outputs both in shapes and numerically.

D. Framework Features

KForge includes several engineering features that make LLM-generated kernel optimization reliable, reproducible, and extensible.

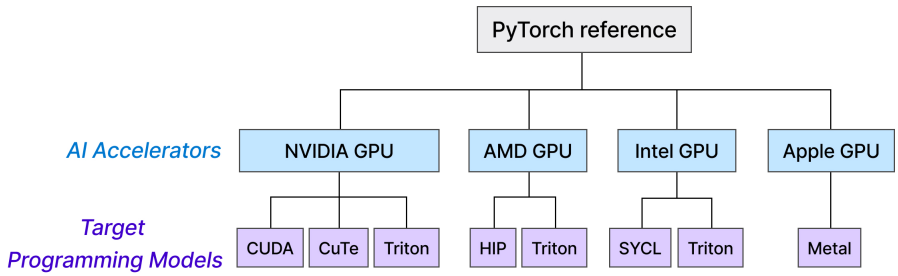

Fig. 2. Given a PyTorch reference, KForge selects and lowers to an appropriate target programming model for each AI accelerator, supporting CUDA, CuTe, Triton, HIP, SYCL, and Metal across NVIDIA, AMD, Intel, and Apple hardware.

- **Candidate guardrails.** KForge supports required and banned code patterns, allowing users to reject candidates that violate constraints. For example, default bans can prevent any undesired changes such as data type down-casting, fast-math flags, or changes to global precision settings.
- **Prompt editing.** KForge lets users override or append to the generation prompt globally or for specific iterations. This makes it possible to inject optimization hints, enforce experiment-specific instructions, or test alternative prompting strategies without modifying the core prompt templates.
- **Auxiliary code injection.** Users can provide auxiliary source files or directories to be included in all iterations or only selected iterations. This is useful for supplying helper kernels, prior implementations or any other reference code.
- **Structured artifacts for reproducibility.** Prompts, model responses, generated candidates, evaluation results, and profiling outputs are saved per model and per iteration, making experiments easier to inspect and reproduce.
- **Custom measurement hooks.** KForge supports user-defined action scripts that consume evaluation results and generated artifacts to compute task-specific measurements or scores, such as end-to-end runtime.

IV. Case Studies

We present two KForge case studies on backends with very different baseline reference availability: NVIDIA, where we compete against years of hand-tuned reference kernels, and Intel Arc, where comparable hand-tuned references are limited. For both cases, KForge is powered by Claude Opus 4.6 in high-effort mode.

A. GPT-OSS Optimization

We target decode-path kernels in the modern MoE-based gpt-oss-20b architecture [17], using TensorRT-LLM v1.3.0rc9 as our baseline. To identify optimization candidates, we profile an inference run with `nsys` and take the top 10 entries from the CUDA GPU Kernel Summary table. These fall into two categories: (1) proprietary kernels distributed only as cubins (e.g., attention and GEMM-with-activation fusions) and (2) kernels distributed with source code. We focus on category (2), since the available reference implementation provides a concrete starting point for optimization, and select three kernels: Fused Add + RMSNorm, MoE finalize, and Bias + RoPE + KV update. The chosen kernels together are components of the attention and MLP modules of the gpt-oss-20b MoE architecture.

In our experiments, we use the `trtllm-bench` script[1] in throughput mode with a workload of 512 requests, 1024-token prefill, 8192-token decode, and batch size 8, running 7 independent iterations of both the baseline and the KForge-optimized model. Initial profiling with `nsys` revealed high variance in time to first token (TTFT), time per output token (TPOT), and end-to-end latency across runs. We traced this to nondeterministic behavior in the autotuner, whose kernel selections varied from run to run. Disabling the autotuner and pinning the GPU clock to 1500 MHz produced reproducible latencies, and we use this configuration for both the baseline and KForge-optimized model.

1) Micro benchmarking: To evaluate kernel performance across a range of workloads, we benchmark each kernel in isolation over a sweep of batch sizes. Results are reported in Table I. Our implementations are consistently faster than the baseline across nearly all configurations. The only exception is the Bias + RoPE + KV update kernel at batch size 8, where KForge is marginally slower; at every larger batch size, the speedup grows with batch size. The MoE finalize kernel shows the most pronounced trend. While the speedup at small batch sizes is modest (1.06 to 1.10×), it increases to 1.43× at batch size 128, indicating that our kernel scales better than the baseline as the workload grows.

2) End-to-end performance evaluation: On NVIDIA B200, integrating the optimized kernels yields a consistent end-to-end performance improvement. The results in Table II are means over 7 independent iterations, with run-to-run coefficient of variation below 0.1% for both metrics, more than an order of magnitude smaller than the reported gains. System output throughput improves by 2.12%, and total wall-clock time decreases by 2.07%. While modest in absolute terms, gains of this size are meaningful against a vendor-optimized runtime that already represents years of engineering effort, and they compound at deployment scale.

[1] https://nvidia.github.io/TensorRT-LLM/commands/trtllm-bench.html

TABLE I
PER-KERNEL EXECUTION TIME (μS) FOR VARIOUS BATCH SIZES ON NVIDIA B200.

Kernel Name	Method	Batch Size						
		1	**4**	**8**	**16**	**32**	**64**	**128**
Fused Add + RMSNorm	Baseline	2.59	2.87	2.86	2.87	2.93	2.96	3.07
	KForge	**2.29**	**2.59**	**2.54**	**2.59**	**2.65**	**2.67**	**2.78**
	Speedup	1.13×	1.11×	1.13×	1.11×	1.11×	1.11×	1.10×
MoE finalize	Baseline	1.85	2.14	2.18	2.25	2.39	2.87	4.06
	KForge	**1.69**	**2.01**	**2.02**	**2.05**	**2.14**	**2.33**	**2.83**
	Speedup	1.09×	1.06×	1.08×	1.10×	1.12×	1.23×	1.43×
Bias + RoPE + KV update	Baseline	3.08	3.14	**3.10**	3.19	3.26	3.30	3.60
	KForge	**3.07**	**3.13**	3.12	**3.12**	**3.24**	**3.22**	**3.42**
	Speedup	1.00×	1.00×	0.99×	1.02×	1.01×	1.02×	1.05×

TABLE II
END-TO-END PERFORMANCE ON NVIDIA B200.

Metric	Baseline	KForge	Δ
Throughput (tok/s)	2601.55	**2656.61**	+2.12%
Wall-clock time (s)	1612.24	**1578.82**	−2.07%

TABLE III
EXECUTION TIME (MS) ON INTEL ARC B580.

Problem	`t.comp`	KForge	Speedup
37_Matmul_Swish_Sum_GN	23.50	**5.73**	4.1×
22_Matmul_Scale_LSE_Mish	9.86	**1.51**	6.5×
88_Gemm_GN_Swish_Mul_Swish	10.00	**1.65**	6.1×
62_Matmul_GN_LReLU_Sum	10.00	**1.83**	5.5×

B. Triton Kernels on Intel Arc B580

We evaluate KForge on a non-NVIDIA platform by generating Triton kernels for the Intel Arc B580 (Battlemage). Our benchmark is the GEMM + tail-ops subset of KernelBench Level 2 (37 problems). Each problem runs five iterations of the generate-refine loop, and we compare the best generated kernel against the faster of two baselines: `torch.compile` and eager-mode PyTorch. KForge achieves a 5.13× geometric mean speedup across the 37 problems.

Table III shows four representative problems spanning the common tail-op patterns in this subset: pointwise activations with reductions (37, 62), normalization fused with elementwise ops (88, 62), and LogSumExp with activation chains (22). We manually inspected the kernels generated for these problems and find that they typically combine four techniques. *Layer-wise fusion* collapses the post-GEMM sequence of operations into a single kernel that reads the matmul output once and writes the result once, eliminating the full-tensor round-trips that dominate baseline runtime. *Mixed-precision execution* runs matmuls in FP16 on the B580's XMX units, with accumulation and post-processing in FP32. *Single-pass reductions* replace two-pass GroupNorm and LogSumExp: small groups fit in registers so mean and variance accumulate alongside the data, while large reductions use the streaming online algorithm similar to FlashAttention's softmax. *Partitioning choices*—sequential per-group loops, 2D group tiles, or per-row streaming—are tuned to group size and shape to maximize occupancy without cross-CTA (Cooperative Thread Array) reductions.

V. LIMITATIONS & FUTURE WORK

Several directions extend KForge beyond the scope of this paper. The current implementation targets PyTorch as the source Deep Learning framework and verifies correctness through numerical tests with datatype-dependent tolerances. Broader framework coverage, such as JAX, and more rigorous verification approaches, such as differential testing across diverse input distributions or formal equivalence checking, would extend the system's applicability and strengthen confidence in LLM-generated kernels.

Beyond these, we highlight three research directions. First, the case studies in this paper rely on a source-level reference implementation to seed the generate-refine loop, but many production kernels, such as fused attention, GEMM-with-activation cubins, and vendor BLAS, ship without source code. Synthesizing kernels from a behavioral specification alone, with correctness checked against the closed-source binary, is a harder regime that we plan to explore. Second, KForge currently produces source-level kernels in programming models such as CUDA C++ and Triton. Targeting a low-level virtual ISA such as PTX would unlock optimizations that source-level programming models cannot express, at the cost of portability and a more challenging synthesis problem. Third, isolated kernel speedups do not always translate to end-to-end gains, since a faster kernel can shift register and shared-memory pressure or perturb the autotuner's selections for neighboring kernels. We aim to extend KForge to jointly optimize kernels within larger architectural components, such as attention or MoE MLP blocks, focusing on the entire block rather than optimizing each kernel independently. Finally, as agents become more capable of long-horizon reasoning, these directions can be further augmented by leveraging the recent advances in autonomous agentic approaches.

VI. Conclusion

We presented KForge, an agentic framework for cross-platform kernel generation and optimization. KForge employs two collaborating LLM-based agents, a generation agent and a performance-analysis agent, to iteratively produce correct and performant kernels across NVIDIA, AMD, Intel, and Apple hardware through six different programming models. Two case studies demonstrate the framework under different baseline reference availability. On NVIDIA B200, KForge produces end-to-end speedup over TensorRT-LLM, a strong baseline for gpt-oss-20b inference. On Intel Arc B580, KForge generates Triton kernels for a backend that lacks a comparable hand-tuned reference. Vendor-optimized runtimes already represent years of engineering effort, and gains in the low single digits compound meaningfully at deployment scale. Agentic kernel synthesis is becoming a practical tool for both production-grade optimization and rapid bring-up on emerging hardware.

References

[1] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski, G. Chauhan, A. Chourdia, W. Constable, A. Desmaison, Z. DeVito, E. Ellison, W. Feng, J. Gong, M. Gschwind, B. Hirsh, S. Huang, K. Kalambarkar, L. Kirsch, M. Lazos, M. Lezcano, Y. Liang, J. Liang, Y. Lu, C. K. Luk, B. Maher, Y. Pan, C. Puhrsch, M. Reso, M. Saroufim, M. Y. Siraichi, H. Suk, S. Zhang, M. Suo, P. Tillet, X. Zhao, E. Wang, K. Zhou, R. Zou, X. Wang, A. Mathews, W. Wen, G. Chanan, P. Wu, and S. Chintala, "Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 929–947. [Online]. Available: https://doi.org/10.1145/3620665.3640366

[2] Apple Inc., "Metal," 2014, graphics and compute API. [Online]. Available: https://developer.apple.com/metal/

[3] Z. Asgar, M. Nguyen, and S. Katti, "Efficient and scalable agentic ai with heterogeneous systems," 2025. [Online]. Available: https://arxiv.org/abs/2507.19635

[4] W. Chen, J. Zhu, Q. Fan, Y. Ma, and A. Zou, "Cuda-llm: Llms can write efficient cuda kernels," 2025. [Online]. Available: https://arxiv.org/abs/2506.09092

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

[6] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," *arXiv preprint arXiv:2205.14135*, 2022.

[7] K. S. Dong, S. Modi, D. Nikiforov, S. Damani, E. Lin, S. K. S. Hari, and C. Kozyrakis, "Kernelblaster: Continuous cross-task cuda optimization via memory-augmented in-context reinforcement learning," 2026. [Online]. Available: https://arxiv.org/abs/2602.14293

[8] C. Hong, S. Bhatia, A. Cheung, and Y. S. Shao, "Autocomp: Llm-driven code optimization for tensor accelerators," in *Machine Learning for Computer Architecture and Systems*, 2025.

[9] Intel Corporation, "Quick Guide to SYCL Implementations," https://www.intel.com/content/www/us/en/developer/articles/technical/quick-guide-to-sycl-implementations.html, accessed Apr. 29, 2026.

[10] R. T. Lange, A. Prasad, Q. Sun, M. Faldor, Y. Tang, and D. Ha, "The ai cuda engineer: Agentic cuda kernel discovery, optimization and composition," *arXiv preprint*, 2025.

[11] E. Lin, S. Modi, S. K. S. Hari, Q. Huang, Z. Ye, N. Qin, F. Zhou, Y. Zhang, J. Wang, S. Damani, D. Peri, O. Xie, A. Kane, M. Maor, M. Behar, T. Cao, R. Mehta, V. Singh, V. S. Mailthody, T. Chen, Z. Ye, H. Chen, T. Chen, V. Grover, W. Chen, W. Liu, E. Chung, L. Ceze, R. Bringmann, C. Zeller, M. Lightstone, C. Kozyrakis, and H. Shi, "Sol-execbench: Speed-of-light benchmarking for real-world gpu kernels against hardware limits," 2026. [Online]. Available: https://arxiv.org/abs/2603.19173

[12] M. Milakov and N. Gimelshein, "Online normalizer calculation for softmax," 2018. [Online]. Available: https://arxiv.org/abs/1805.02867

[13] A. Modarressi, H. Deilamsalehy, F. Dernoncourt, T. Bui, R. A. Rossi, S. Yoon, and H. Schütze, "Nolima: Long-context evaluation beyond literal matching," 2025. [Online]. Available: https://arxiv.org/abs/2502.05167

[14] NVIDIA, "Cuda toolkit documentation," https://docs.nvidia.com/cuda/.

[15] NVIDIA, "Tensorrt-llm," 2023, large Language Model inference optimization library for NVIDIA GPUs. [Online]. Available: https://github.com/NVIDIA/TensorRT-LLM

[16] NVIDIA Corporation, "CuTe DSL," https://docs.nvidia.com/cutlass/latest/media/docs/pythonDSL/cute_dsl.html, 2026, nVIDIA CUTLASS Documentation. Last updated Apr. 8, 2026. Accessed Apr. 29, 2026.

[17] OpenAI, S. Agarwal, L. Ahmad, J. Ai, S. Altman, A. Applebaum, E. Arbus, R. K. Arora, Y. Bai, B. Baker, H. Bao, B. Barak, A. Bennett, T. Bertao, N. Brett, E. Brevdo, G. Brockman, S. Bubeck, C. Chang, K. Chen, M. Chen, E. Cheung, A. Clark, D. Cook, M. Dukhan, C. Dvorak, K. Fives, V. Fomenko, T. Garipov, K. Georgiev, M. Glaese, T. Gogineni, A. Goucher, L. Gross, K. G. Guzman, J. Hallman, J. Hehir, J. Heidecke, A. Helyar, H. Hu, R. Huet, J. Huh, S. Jain, Z. Johnson, C. Koch, I. Kofman, D. Kundel, J. Kwon, V. Kyrylov, E. Y. Le, G. Leclerc, J. P. Lennon, S. Lessans, M. Lezcano-Casado, Y. Li, Z. Li, J. Lin, J. Liss, Lily, Liu, J. Liu, K. Lu, C. Lu, Z. Martinovic, L. McCallum, J. McGrath, S. McKinney, A. McLaughlin, S. Mei, S. Mostovoy, T. Mu, G. Myles, A. Neitz, A. Nichol, J. Pachocki, A. Paino, D. Palmie, A. Pantuliano, G. Parascandolo, J. Park, L. Pathak, C. Paz, L. Peran, D. Pimenov, M. Pokrass, E. Proehl, H. Qiu, G. Raila, F. Raso, H. Ren, K. Richardson, D. Robinson, B. Rotsted, H. Salman, S. Sanjeev, M. Schwarzer, D. Sculley, H. Sikchi, K. Simon, K. Singhal, Y. Song, D. Stuckey, Z. Sun, P. Tillet, S. Toizer, F. Tsimpourlas, N. Vyas, E. Wallace, X. Wang, M. Wang, O. Watkins, K. Weil, A. Wendling, K. Whinnery, C. Whitney, H. Wong, L. Yang, Y. Yang, M. Yasunaga, K. Ying, W. Zaremba, W. Zhan, C. Zhang, B. Zhang, E. Zhang, and S. Zhao, "gpt-oss-120b & gpt-oss-20b model card," 2025. [Online]. Available: https://arxiv.org/abs/2508.10925

[18] A. Ouyang, S. Guo, S. Arora, A. L. Zhang, W. Hu, C. Ré, and A. Mirhoseini, "Kernelbench: Can llms write efficient gpu kernels?" 2025. [Online]. Available: https://arxiv.org/abs/2502.10517

[19] P. Tillet, H. T. Kung, and D. Cox, "Triton: an intermediate language and compiler for tiled neural network computations," ser. MAPL 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 10–19. [Online]. Available: https://doi.org/10.1145/3315508.3329973