

Towards Agentic LLMs for Hardware-Aware Kernel Generation

Aksara Bayyapu

CMU-CS-25-128

August 20, 2025

Computer Science Department
School of Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:

Tianqi Chen

Zhihao Jia

*Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science.*

Keywords: Large Language Models, Agents, Code Generation, Kernels

To my family and friends, for their constant love and encouragement. To my peers and academic advisors at Carnegie Mellon, for their guidance and collaboration.

Abstract

Recent advances in large language models have furthered the development of agentic systems: pipelines that interleave planning, execution, and iterative refinement. The following thesis demonstrates the design, implementation, and evaluation of such agentic systems so they can be leveraged in underutilized ways, from productivity tasks to refining kernel code. First, we propose Web LLM Agent, which showcases a general-purpose Chrome extension agent that integrates LLMs with diverse web APIs. Beyond constructing a web agent, we aimed to extend agentic systems to the more specialized and impactful domains, such as creating kernels through an LLM Kernel Agent. This agent embodies an iterative decision-making agent that generates, benchmarks, and refines kernels, illustrating the core principles of agent architecture, feedback loops, and sequential optimization. Findings will show that there is a lack of thorough benchmarking and evaluation metrics for kernels, making iterative, generative improvement difficult. The third component of the following paper focuses on creating a lightweight, extensible benchmarking suite, FlashInfer Bench, which resolves this issue. The suite captures execution traces to systematically evaluate and compare low-level kernel implementations for performance and correctness in model inference workloads. Thus, FlashInfer Bench proposes a self-improving, community-driven platform for developing and deploying hardware-aware, high-performance kernels. Together, these contributions emphasize the central importance of agent design, tool orchestration, and integrated feedback loops in unlocking autonomous, high-performance agents.

Acknowledgments

I would like to thank my advisor, Tianqi Chen, for his guidance and support throughout this project. My sincere thanks also go to all the members of Catalyst Lab—especially Shanli Xing, Yiyang Zhai, Alexander Jiang, and Yixin Dong—for their contributions and collaborative spirit. Finally, I extend my appreciation to Zhihao Jia for his support and for serving on my thesis committee.

Contents

- 1 Introduction** **1**
 - 1.1 Background 1
 - 1.2 Motivation 2
 - 1.3 Problem Statement & Scope 3
 - 1.4 Contributions 3
 - 1.5 Summary 4

- 2 Related Work** **5**
 - 2.1 Existing Agents 5
 - 2.2 KernelBench 6

- 3 Web LLM Agent** **9**
 - 3.1 Motivation & Web Agent Criteria 9
 - 3.2 Implementation 9
 - 3.3 Evaluation & User Experience 11
 - 3.4 Web Agent Limitations & Scope 11

- 4 LLM Kernel Agent** **13**
 - 4.1 Motivation & Agentic Decision Frameworks 13
 - 4.2 Agent Architecture and Workflow 14
 - 4.3 Optimization Limitations 15

- 5 FlashInfer Benchmark** **17**
 - 5.1 Motivation 17
 - 5.2 Implementation 18
 - 5.3 How this Sets the Stage for Future Agent Integration 20

- 6 Results and Discussion** **23**
 - 6.1 Synthesis & Cross-Cutting Insights 23
 - 6.2 Common Architectural Motifs 24
 - 6.3 Recurring Bottlenecks & Pivot Points 24

- 7 Conclusion** **27**

- Bibliography** **29**

List of Figures

3.1	High-level architecture of Web LLM Assistant	10
4.1	High-level architecture of LLM Kernel Agent	14
5.1	Example of a <code>Definition</code> schema for matrix multiplication.	19
5.2	Example of a <code>Solution</code> schema generated by an agent.	19
5.3	Example of a <code>Trace</code> schema recording a benchmark result.	20

Chapter 1

Introduction

1.1 Background

Recent advances in machine learning systems have given rise to large language models (LLMs) capable of generating text, writing code, and complex reasoning . Pioneering systems such as GPT-3 demonstrated that scaling model parameters and pretraining data could result in few-shot generalization across dozens of natural language processing (NLP) tasks. Building on these capabilities, researchers have begun to embed LLMs within agentic frameworks, or software systems that leverage an LLM’s generative power to combine planning, self-evaluation, and iterative refinement to accomplish complex tasks autonomously. Agentic systems layer decision-making loops atop the model’s outputs, enabling workflows that interleave reasoning steps with tool invocations such as API calls, code execution, and retrieval. These pipelines have shown promise across numerous domains in academia and industry.

Agentic pipelines augment a base LLM with external tools—such as code execution environments, profilers, and retrieval systems—to create closed-loop workflows. A typical agentic loop proceeds by (1) formulating a plan or generating code, (2) executing or querying a tool, (3) analyzing the results, and (4) refining the plan or code based on analysis. This paradigm has enabled impressive applications, including automated spreadsheet manipulation, multi-step question answering, and end-to-end research prototyping. Crucially, effective agentic systems rely on two pillars: (i) robust integration with external tools, and (ii) reliable feedback channels that inform the agent’s next steps. These insights motivate our investigation into how similar loops might guide kernel optimizers.

The prospect of using LLMs as code assistants has captured broad interest in both academia and industry. Early work showed that LLMs could generate boilerplate code, translate between programming languages, and suggest API usage. More recent efforts integrate LLMs into IDE plugins and chat interfaces that allow developers to describe desired functionality in natural language and receive complete method stubs or data-analysis notebooks [8]. However, these systems typically focus on high-level logic rather than low-level, performance-critical code. Utilizing LLM’s to generate systems that create performance-critical code is a domain that is yet to be thoroughly explored.

Machine learning (ML) systems are highly based on the efficient execution of computation-

ally intensive operations such as matrix multiplications, convolutions, reductions, and non-linear functions. These operations are the foundational building blocks of deep learning architectures. They are used in convolutional neural networks (CNNs), transformers, and graph neural networks. The execution of these operations is designated to Graphics Processing Units (GPUs).

The GPU is uniquely suited for data-parallel computations, enabling efficient execution of computationally intensive operations. A GPU consists of dozens to thousands of compute cores organized into streaming multiprocessors (SMs), each with its own register file, shared memory, and instruction scheduler. High performance on GPU demands careful management of on-chip resources (registers, shared memory, cache) and occupancy, as well as coalesced memory accesses and latency hiding via fine-grained thread scheduling.

A GPU kernel is a function written specifically for running on the GPU, and is typically executed in parallel by thousands of lightweight threads. GPU kernels are different from most functions because they implement low-level logic for ML operations. The performance of GPU kernels directly impacts the throughput, latency, and energy efficiency of the training and inference pipelines. Given the increasing usage of modern large language models and datasets, even the smallest improvements in kernel performance can translate to considerable savings in compute time and energy consumption.

High-performance machine learning frameworks, such as PyTorch and TensorFlow, abstract away the low-level kernels from the end user for the sake of simplicity and clarity. Despite this abstraction, the frameworks rely on highly optimized CUDA kernels under the hood, provided by libraries such as cuDNN, CUTLASS, and TensorRT, which are meticulously fine-tuned for various GPU architectures and workloads [3, 5, 6, 9]. Fine-tuning kernels to optimize them for their machines is a time-intensive process that requires the knowledge of experts. Consequently, the optimization process presents a bottleneck as new ML architectures and hardware accelerators emerge.

1.2 Motivation

Recent advances in large language models (LLMs) have opened new possibilities for creating artificially intelligent (AI) agents, software systems that combine planning, self-evaluation, and iterative refinement to accomplish complex tasks autonomously [2, 7]. The development of agents is promising, appearing in domains such as code synthesis and dynamic decision-making. Despite the enormous effort being put into the development of agent pipelines, a significant gap remains at the intersection of LLM-driven agents and high-performance computing. Specifically, leveraging agents to generate and optimize GPU kernels. Kernels, or functions written specifically for running on the GPU, are critical components for the type of high-performance computing necessary to meet the demands of modern machine learning workloads.

As the complexity and scale of ML workloads has grown, so has the need for adaptable and efficient kernel generation strategies. Traditionally, expert programmers spend significant effort manually writing and tuning CUDA kernels to squeeze out maximum performance on specific GPU architectures. This process is not only time-consuming but also error-prone and brittle across models, workloads, and hardware generations. Furthermore, the pace at which new ML models and specialized hardware accelerators are introduced makes this manual optimization

loop increasingly unsustainable.

One solution to address this growing demand is to use large language models (LLMs) to create the kernels. LLMs that have been trained on code, including GPU kernels, should theoretically be able to generate custom CUDA kernels. Furthermore, LLMs have knowledge of architectures and specific workloads, making them capable of effectively optimizing kernels for their usage. However, a crucial component of their success in generating correct and performant kernels depends on our ability to evaluate and benchmark the code with domain-specific feedback.

When an LLM proposes a kernel implementation, its suggestions often lack the micro-optimizations and hardware-aware idioms necessary for peak GPU throughput. Bridging this gap requires not only more structured prompting and fine-tuning, but also a feedback mechanism that ties generated code back to real performance measurements.

1.3 Problem Statement & Scope

The following paper will demonstrate a stepwise exploration of three interconnect components that will eventually address the aforementioned gap. It will investigate how to harness LLM-based agents both as general productivity tools and as domain-specialized optimizers for GPU code. It begins with an exploratory prototype that demonstrates seamless integration of an LLM with rich external APIs (calendar, email, music, document synthesis). Building on the lessons from that prototype, we then develop a Proof-of-Concept Kernel Agent that frames GPU-kernel synthesis as a sequential decision problem and iteratively refines generated code via benchmarking feedback. Lastly, we construct a Kernel Benchmarking Framework, enabling a robust infrastructure for end-to-end evaluation and evolutionary refinement of kernels, laying the groundwork for truly autonomous, performance-driven agents.

1.4 Contributions

The following details the three components of the stepwise exploration.

Web LLM Agent

MLC Assistant is an interactive LLM chat agent tool built as a chrome extension that helps users write papers, synthesize notes, schedule events in Google Calendar, draft emails in Gmail, play music on Spotify, and more. It overcomes challenges in API orchestration, state management, and user feedback collection that inform the design of specialized agentic loops.

LLM Kernel Agent

The Kernel Agent works as an autonomous loop that treats CUDA-kernel optimization as a sequential decision process: propose code, benchmark on target hardware, analyze performance bottlenecks, and instruct the LLM to revise its implementation—demonstrating feasibility on small workloads and identifying key scalability hurdles.

Flashinfer Benchmark

A scalable system for realistic workload construction, fine-grained metric collection, and population-based search operators that evolve high-performance GPU kernels. Although not

itself an agent, this framework provides the essential evaluation backbone for future agentic optimizers.

Together, these contributions demonstrate that building effective agentic systems for GPU kernel optimization hinges not only on clever synthesis loops, but also on a solid benchmarking backbone that can faithfully evaluate and guide future agents toward high-performance implementations.

1.5 Summary

The remainder of the thesis is structured as follows:

Chapter 2 reviews background on LLMs, agentic systems, GPU architecture, and prior work in code synthesis and benchmarking.

Chapter 3 details the design, implementation, and evaluation of the MLC Assistant, including its interactive workflow and lessons learned.

Chapter 4 presents the LLM Kernel Agent, formalizing kernel optimization as a sequential decision problem and analyzing its performance on prototypical workloads.

Chapter 5 describes the GPU Kernel Benchmarking Framework, its architecture, evolutionary operators, and case studies across diverse kernels.

Chapter 6 synthesizes cross-cutting insights, discusses trade-offs between interactivity, autonomy, and infrastructure, and outlines a vision for fully agentic optimizers.

Chapter 7 concludes with a summary of contributions and a roadmap for future research.

Appendices provide experimental details, additional benchmarks, and code listings.

Chapter 2

Related Work

2.1 Existing Agents

Several recent projects have explored the design and evaluation of large language model–driven agents that operate in interactive environments. These systems provide valuable case studies for how agents can be constructed, as well as how their performance can be systematically measured. The following section introduces some web agents relevant to the work of the first exploratory project.

Qwen-VL

Qwen-VL extends agentic systems into the vision–language domain. It equips agents with the ability to process and act on multimodal inputs. This includes interpreting images in the context of text queries or grounding instructions in visual content. Alongside model capabilities, Qwen-VL introduced benchmarks that test vision-language reasoning and prompting strategies, offering insights into how multimodal agents can integrate perception with decision-making [1].

WebVoyager

WebVoyager investigates agents designed for web-based tasks. It provides structured benchmarks that span common digital workflows, including scheduling events in calendars, drafting and sending emails, and retrieving information from documents. The benchmark tasks emphasize multi-step reasoning and interaction, illustrating how an LLM can be embedded in an agentic loop that sequences actions across real applications [4].

browser-use

The browser-use framework focuses on standardized web navigation and information-processing tasks. It highlights how agents, guided by models such as GPT-4o, can be evaluated under controlled conditions with defined prompts and actions. In particular, it emphasizes the role of prompt patterns in shaping agent robustness and provides a set of repeatable benchmark tasks for studying decision-making in constrained web settings [12].

UI-TARS

UI-TARS shifts attention to user interface navigation. It defines tasks where agents must complete GUI-driven operations, such as filling forms or performing structured actions within software applications. Importantly, UI-TARS introduces human–agent comparisons, assessing not only whether the agent can finish a task but also how its efficiency and error recovery compare

to human performance [10].

Benchmarking

Alongside the development of agentic systems, researchers have placed increasing emphasis on building benchmarks to measure their effectiveness. Unlike static language-model evaluations, which assess accuracy on fixed datasets, agent benchmarks are designed to capture the dynamic, multi-step nature of interaction. They evaluate not only whether an agent produces a correct final answer, but also how it sequences actions, recovers from errors, and adapts to intermediate feedback.

In developing our own Web LLM Agent, we initially sought to replicate or contribute useful benchmark metrics in this space. However, we found a clear mismatch: state-of-the-art systems such as *browser-use* are vision-based and supported by large-scale datasets spanning Amazon, Google Maps, and complex multistep workflows. By contrast, our agent was strictly text-only and vision-less, which limited its precision in benchmarks designed around multimodal perception. We aimed to create a benchmark better suited for text-based web agents.

However, scaling such a system without vision proved infeasible and misaligned with our lab’s strengths. This limitation motivated us to redirect our focus toward domains where text-based reasoning remains central and benchmarking is underdeveloped. One such domain is high-performance computing, specifically GPU kernel generation and optimization. Unlike web navigation, kernel synthesis tasks are inherently textual: kernels are written in CUDA or similar programming languages, and their quality can be assessed through compilation, execution, and performance measurement rather than visual grounding.

2.2 KernelBench

KernelBench emerges as an alternative benchmarking direction. Rather than evaluating navigation or multimodal perception, it provides a framework for assessing how agents generate, refine, and optimize GPU kernels under realistic workloads. The emphasis is on:

- Workload construction: defining representative computational kernels (e.g., matrix multiplications, reductions, convolutions) that reflect common ML operations.
- Performance measurement: collecting fine-grained metrics such as throughput, latency, memory utilization, and occupancy.
- Iterative refinement: enabling agents to propose code, run benchmarks, analyze bottlenecks, and revise implementations in a loop.

Key features include:

- A diverse benchmark suite of 250 tasks, derived from PyTorch-based neural network workloads. These tasks span:
 - Level 1 (100 tasks): Fundamental operations like convolutions, matrix multiplications, and layer normalization.
 - Level 2 (100 tasks): Simple fused patterns such as Conv + Bias + ReLU.
 - Level 3 (50 tasks): Full sub-models or architectures (e.g., MobileNet, VGG).

- A dual-challenge evaluation framework centered on correctness, via functional equivalence to reference PyTorch implementations, and performance, via speedup relative to the baseline.

By centering on text-only tasks with objective, measurable outcomes, KernelBench avoids the limitations we encountered in multimodal web-agent evaluation. At the same time, it addresses a critical gap: while vision–language agents have well-established benchmarks, high-performance kernel generation has no analogous infrastructure. KernelBench fills this void by offering a domain-specific backbone for future LLM-based optimizers [11].

Chapter 3

Web LLM Agent

3.1 Motivation & Web Agent Criteria

The Web LLM Agent was designed as a privacy-preserving, on-device AI assistant embedded within a Chrome extension, leveraging the WebLLM inference engine. This approach reflects a shift from cloud-based models that sometimes require tens of billions of parameters to efficient, specialized models optimized for execution in the browser. Thus, this approach explores the feasibility of building an in-browser LLM assistant that could operate entirely on-device. As a result, users gain enhanced privacy, reduced latency, and zero dependency on external APIs. The web environment enables a broad reach (installation-free), rich tasks, and portability via modern runtimes like WebGPU. With advances in lightweight models and the availability of WebGPU, it became possible to envision agents that no longer depend on cloud infrastructure, offering both privacy and accessibility.

Chrome extensions provide a natural deployment environment since the web is ubiquitous, task-rich, and requires no installation barriers. Additionally, browser APIs expose rich interactions with content through the DOM, tabs, and network requests. Lastly, Service workers and content scripts make it possible to structure agent workflows.

Prior work on agentic systems has primarily focused on API orchestration. Existing web-agents, such as those discussed in the aforementioned section, demonstrate scheduling, email management, and content generation across productivity platforms. These systems are cloud-hosted and rely heavily on REST APIs for task execution.

A persistent challenge has been the hosting of LLMs. Running even 7B-scale models in the browser are computationally intensive. WebLLM provided a starting point, but constraints in memory, compilation, and latency framed the problem of creating a lightweight yet general-purpose agent.

3.2 Implementation

The Web LLM Agent was implemented as a Chrome extension consisting of three layers. Firstly, the WebLLM delivers high-performance, in-browser LLM inference, powered by WebGPU and WebAssembly, offering near-native performance by compiling model kernels to GPU-executable

code. It provides full compatibility with OpenAI API paradigms, including streaming outputs, JSON-mode, and (in progress) function calling which makes integration seamless for web agents.

Second, is the MLC Assistant, or the front-end agent interface. It consists of a popup UI: when a user clicks the extension icon, the popup provides the interactive prompt interface. Within this component, the background service worker manages global model state and orchestrates communication across components. Finally, content scripts interact with page DOM, invoke tools, send and receive messages to and from background. All of these components are connected via Chromes asynchronous message passing infrastructure.

The third and final component is the Web Agent Interface: a tool abstraction layer providing tools for page functionality like retrieving selected text or replacing content. It also supports higher-level editing on platforms such as Overleaf, Google Docs, Spotify, and Google Calendar. The agent delegates to WAI for DOM ops. If specific actions require APIs, WAI handles them transparently.

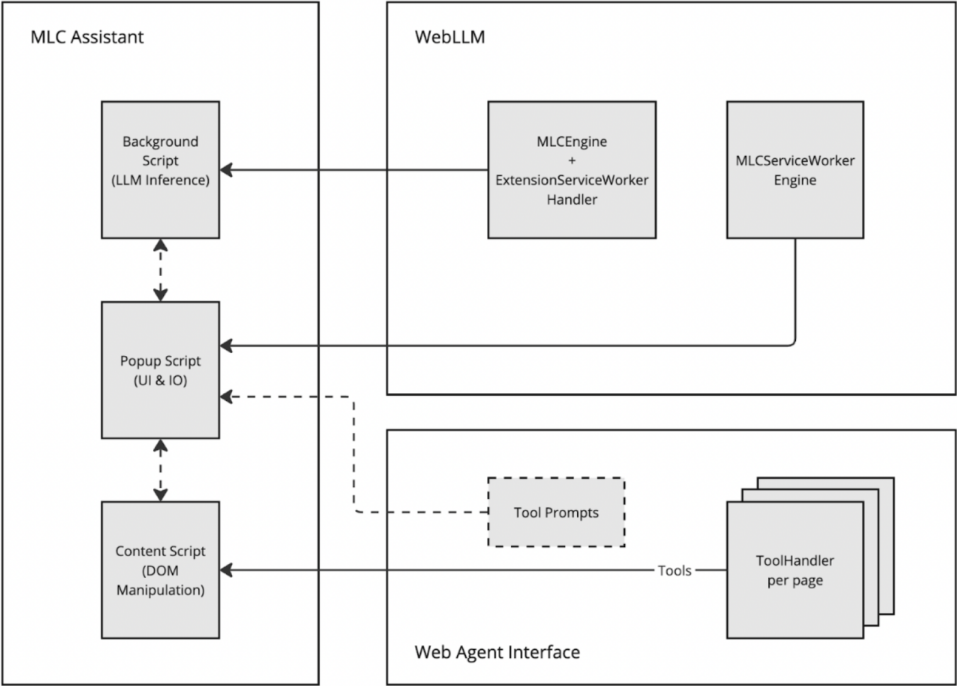


Figure 3.1: High-level architecture of Web LLM Assistant

Additional design considerations included prompting strategies to ensure content retention with minimal memory usage. We also considered state management and ways to persist conversation across tabs and sessions. The main priority was building out tool integration through WAI to ensure structured access to Google Calendar, Gmail, Spotify, and online editors (Overleaf, Google Docs).

3.3 Evaluation & User Experience

Qualitatively, the system demonstrated that in-browser agents were feasible, though limited in scope. Users could issue queries like “schedule a meeting on Tuesday at 2PM” or “play a specific playlist on Spotify” and observe the assistant orchestrate API or DOM-level actions directly.

The user flow would begin with the user invoking the pop-up and inputting their prompt. The background service worker receives the prompt and manages the sessions. The ServiceWorkerMLCEngine (within the worker thread) compiles and runs the model, returning streaming responses. Content scripts trigger tool actions (e.g., insert text, call APIs) through WAI if required. Responses are streamed back to the pop-up, completing the interaction cycle.

The advantage of creating this entirely local inference via WebLLM is the preservation of user data and the reduction of reliance on cloud infrastructure. The separation between front-end (UI), engine inference, and WAI tools enables scalable, maintainable development. The real-time tool invocation and response streaming establish a powerful agentic feedback loop on-device. Finally, leveraging WebGPU and WebAssembly, the Web LLM Agent runs on diverse consumer hardware.

The streaming output from WebLLM provided real-time interactivity; however, performance was constrained by factors such as model size and browser memory limits, as well as compilation times on initial load and latency compared to server-hosted assistants.

These constraints shaped both the user experience and the types of workflow that the agent could realistically support.

3.4 Web Agent Limitations & Scope

The Web LLM Agent highlighted the promise of in-browser agents, and several limitations became apparent. Notably, existing web agent benchmarks assume multimodal capabilities (e.g., vision-based systems trained on Amazon or Google Maps tasks). Our text-only agent lacked competitive precision in such settings. Without vision and large-scale datasets, our approach could not scale to state-of-the-art performance. Further work on text-only browser agents did not align with our research focus, motivating a pivot.

The project validated the feasibility of LLM agents in the browser, but also exposed the fundamental gap between text-only prototypes and state-of-the-art multimodal web agents. This recognition led us to pivot to kernel benchmarking and model evaluation, where our lab’s expertise could more directly advance the field.

Chapter 4

LLM Kernel Agent

4.1 Motivation & Agentic Decision Frameworks

Despite the improvements in LLMs’ ability to write code, a fundamental gap remains in their ability to produce performant low-level GPU kernels. Writing GPU kernels is notoriously difficult, requiring specialized expertise in parallel programming, GPU architecture, and hardware-aware optimizations. For many machine learning workloads, these kernels represent a bottleneck: they are essential for performance but costly and time-consuming to develop by hand. As a result, there is strong motivation to design agentic systems that can automate kernel generation and refinement.

Agentic decision frameworks provide a natural paradigm for this problem. Unlike single-shot code generation, agents interleave planning, execution, and iterative refinement, enabling a form of feedback-driven search. By embedding kernel generation in this framework, LLMs can evolve initial drafts of GPU code into increasingly optimized implementations through repeated evaluation and correction. This reframes kernel authoring as a sequential decision-making process, where each iteration produces new information that guides the next.

KernelBench, an open-source benchmark suite from Stanford, provides a key stepping stone in this direction. It formalizes the evaluation of LLM-generated kernels by measuring whether they compile, whether their outputs match a trusted PyTorch baseline, and how their runtime performance compares. While KernelBench itself is a static benchmark focused on one-off evaluation, it is explicitly designed to support agentic workflows. By turning correctness and performance metrics into feedback signals, KernelBench enables the creation of agents that do more than generate kernels: they learn to iteratively improve them [11]. This inspired the design of our LLM Kernel Agent, which treats kernel optimization not as a single-shot generation problem but as a structured feedback loop.

In summary, the motivation for the LLM Kernel Agent is twofold: (1) to address the difficulty and bottleneck of expert-written GPU kernels, and (2) to demonstrate how agentic decision frameworks can open the door to autonomous kernel synthesis and optimization.

4.2 Agent Architecture and Workflow

The LLM Kernel Agent was designed around an iterative feedback loop. At a high level, the system follows a generate–compile–evaluate–refine cycle, where each stage provides signals that guide the model toward more correct and performant kernels. This agentic framing distinguishes the system from static kernel generation and enables continuous improvement through structured decision-making. The architecture is composed of four primary components:

Generation Module: The agent begins by prompting an LLM to produce an initial candidate kernel in CUDA. Prompts include task descriptions, reference PyTorch implementations, and optional hints about constraints (e.g., memory layout, reduction patterns).

Compilation & Sanity Checks: Candidate kernels are compiled with toolchains such as `nvcc`. This step filters out syntax errors and ensures that the generated code is well-formed before execution.

Execution Harness: Correctness is verified by running the kernel against unit tests and comparing its outputs to a trusted PyTorch baseline. In addition, the agent records execution time, memory usage, and throughput metrics, producing a quantitative profile of each kernel.

Feedback & Refinement: Results from compilation and execution are translated into structured feedback that is returned to the LLM. This creates a closed feedback loop that operationalizes the “trial-and-error” process of optimization.

The workflow proceeds iteratively:

1. The agent receives a kernel task specification (e.g., “implement layer normalization”).
2. The generation module produces an initial implementation.
3. The candidate is compiled and executed; results are logged against correctness and performance metrics.
4. Feedback is formatted into natural language or structured system messages and returned to the LLM.
5. The LLM proposes a revised kernel, informed by the results of prior attempts.
6. Steps (2)–(5) repeat until convergence criteria are met, such as correctness achieved within tolerance or performance surpassing a baseline threshold.

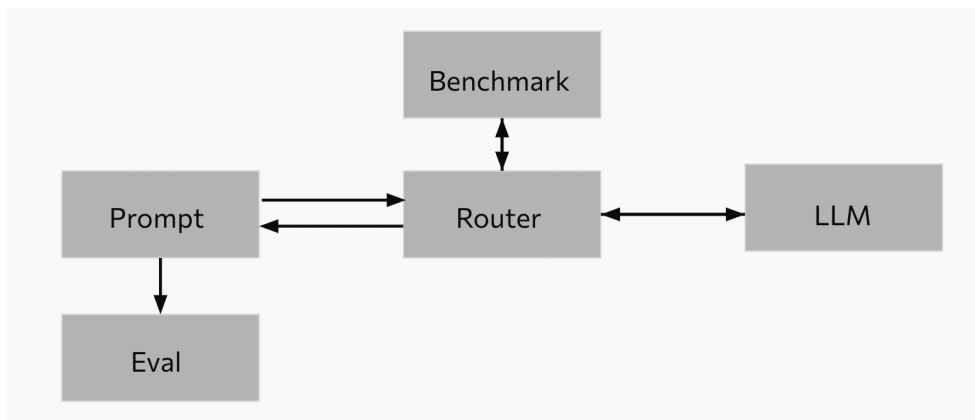


Figure 4.1: High-level architecture of LLM Kernel Agent

Next, we discuss the benchmarking tools and Model Context Protocol (MCP) servers integrated into the LLM Kernel Agent. These servers act as callable services within the agent loop, exposing evaluation capabilities as modular interfaces.

evaluate_kernel: This server wraps the KernelBench evaluation suite. It accepts a generated kernel and a reference baseline, then benchmarks across three key dimensions: the ability to compile, correctness of outputs, and runtime performance. This provides a lightweight yet robust signal for iterative improvement.

nsys_profiler: This server compiles and executes a generated CUDA kernel similarly to `evaluate_kernel`, but augments evaluation with NVIDIA Nsight Systems (`nsys`) profiling. It records detailed performance statistics such as total runtime per kernel, number of kernel calls, and average execution time. These statistics support fine-grained analysis of GPU behavior and enable optimization strategies that go beyond correctness into hardware efficiency.

Two principles guided the agent’s architecture. First, *modularity*: each component—generation, compilation, execution, feedback—is designed to be swappable, enabling experimentation with different LLMs, compilers, or feedback strategies. Second, *incrementality*: the workflow is explicitly iterative, encouraging small, interpretable changes across cycles rather than wholesale regeneration. This mirrors the workflow of human programmers and maximizes the utility of feedback signals.

In summary, the agent architecture operationalizes kernel optimization as an iterative decision process, embedding the LLM within a structured environment of tools and feedback. This design not only improves correctness and performance over naive generation but also creates a generalizable framework for autonomous GPU code authoring.

4.3 Optimization Limitations

While the LLM Kernel Agent was able to generate and iteratively refine GPU kernels, several structural limitations emerged that reduced the reproducibility, comparability, and ultimate effectiveness of the system.

A major limitation was the absence of standardized input workloads across contributors and experiments. Kernels were often tested with ad-hoc or randomly chosen input shapes, which were neither logged nor versioned. This lack of specification meant that evaluation results could not be reproduced consistently, even on the same hardware. Without a canonical input set, performance comparisons across models, runs, or backends were unreliable.

Although the evaluation tools (`evaluate_kernel`, `nsys_profiler`) provided correctness and performance metrics, there was no unified evaluation protocol to tie them together. In particular, no standard input specification for correctness and performance testing. Furthermore, there was no persistent record of input generation or evaluation seeds. Lastly, no consistency in which benchmarks were considered “passing.”

This fragmented approach made it difficult to verify results, compare them across contributors, or accumulate evidence over time.

The system lacked a persistent trace format to log kernel generation. Essential metadata (such

as the kernel source code, the LLM model and parameters used to generate it, hardware and environment configuration, and collected performance statistics) were all stored in isolation, if at all. As a result, performance results could not be tied back to a specific kernel lineage or compared systematically across agents and backends. Without persistent traces, agent generations were effectively ephemeral and could not be reused or aggregated for broader benchmarking.

Both our benchmark and KernelBench shared an important structural limitation: they primarily evaluated LLM-generated kernels against each other, while excluding expert-written baselines. In practice, most high-performance GPU kernels are human-written and hand-optimized. By not including these kernels in the benchmarking process, the evaluation risked being self-referential. In other words, agents were compared to themselves, rather than against state-of-the-art human-optimized kernels that represent real-world performance ceilings. This limited the ability to assess whether LLM-generated kernels were competitive in practice.

These limitations highlighted the need for a stronger evaluation backbone. We needed a standardized, traceable, and inclusive of human-optimized kernel to fully support agentic kernel generation research, prompting the next research project.

Chapter 5

FlashInfer Benchmark

5.1 Motivation

The development of the FlashInfer Benchmark was driven by a simple but critical realization: without robust, reproducible benchmarking, there is no reliable way to measure progress in agentic GPU kernel generation. While our earlier prototypes could generate, refine, and even execute kernels, they lacked the standardized infrastructure needed to compare solutions across contributors, models, or time. This meant that kernels existed only as isolated experiments, making it impossible to track long-term improvement or meaningfully evaluate the effectiveness of agent-driven approaches.

Robust benchmarking matters for agents because it provides the feedback loop that makes iterative refinement possible. Just as large language models rely on reinforcement learning from human feedback (RLHF) to improve their outputs, kernel-generation agents require structured signals on correctness, performance, and reproducibility. Without this grounding, agents may generate syntactically valid but inefficient or incorrect code, meaning results that cannot be verified, compared, or reused.

The lack of a standardized workload specification was one of the largest barriers. Input shapes were often generated arbitrarily, correctness tests varied across runs, and results were not logged in a reproducible format. This fragmented process not only blocked comparisons between agent outputs but also prevented benchmarking against human-authored baselines, which is the true performance ceiling in GPU kernel optimization.

FlashInfer Benchmark was created to fill this gap by offering a unified, extensible framework for defining workloads, executing benchmarks, and analyzing results. Its design emphasizes three principles:

- **Reproducibility:** Standardized schemas for workloads (`Definition`), implementations (`Solution`), and results (`Trace`) ensure that every benchmark can be recreated and verified.
- **Comparability:** Results from agents, humans, and compilers are logged in a common format, enabling fair evaluation across implementations.
- **Evolvability:** Rather than being a one-time testbed, FlashInfer Benchmark is designed as a community-driven ecosystem—capable of growing alongside new workloads, new

models, and new agentic strategies.

In short, the motivation for FlashInfer Benchmark was not simply to measure performance once, but to create the foundation for continuous, structured improvement in kernel generation. By capturing correctness, performance, and provenance in a reproducible manner, it transforms kernel optimization from a fragmented research effort into a transparent, collective endeavor.

5.2 Implementation

FlashInfer Bench is built around two key runtime objects that structure the execution and analytics workflow for benchmarking kernels:

- **Benchmark (Execution Engine):** Loads definitions (`Definition`), implementations (`Solution`), and configs. Builds reference and candidate kernels into callable functions. Executes benchmarks and logs results as immutable `Trace` objects.
- **TraceSet (Analytics Layer):** Aggregates `Trace` results across kernels, workloads, and devices. Supports merging, exporting, visualization, and top-kernel selection.

This design ensures that every workload, implementation, and benchmark result is self-describing, language-agnostic, and easily loadable by the `Benchmark` and `TraceSet` APIs.

The Python package provides the tooling to contribute to the benchmark suite and is supplemented by a lightweight command-line interface:

- `flashinfer-bench run` — structured benchmarking execution.
- `flashinfer-bench report` — merging, summarizing, and visualizing traces.

Each benchmark run is structured around three schema-driven components. These schemas provide a consistent format that ensures results are reproducible, language-agnostic, and easily loadable by both the `Benchmark` and `TraceSet` APIs.

Definition

The `Definition` schema specifies the workload. It captures the input and output tensor formats, the semantics of dimensions, and provides a mathematical reference implementation in PyTorch. This ensures that every workload is unambiguously described and can be validated across implementations.

```

{
  "name": "matmul_bf16",
  "inputs": [
    {"name": "A", "dtype": "bfloat16", "shape": ["M", "K"]},
    {"name": "B", "dtype": "bfloat16", "shape": ["K", "N"]}
  ],
  "outputs": [
    {"name": "C", "dtype": "bfloat16", "shape": ["M", "N"]}
  ],
  "reference": "torch.matmul(A, B)"
}

```

Figure 5.1: Example of a `Definition` schema for matrix multiplication.

Solution

The `Solution` schema represents an implementation of a workload. It links back to a `Definition` by name and records the implementation details, including the author (human or agent), programming language, and entry point for benchmarking. Solutions can be written in any language (CUDA, Triton, etc.) and are validated against the reference.

```

{
  "definition": "matmul_bf16",
  "author": "agent_llama3",
  "language": "triton",
  "entry_point": "matmul_kernel.py::matmul",
  "metadata": {
    "generated_by": "LLaMA 3.1-8B",
    "date": "2025-06-15"
  }
}

```

Figure 5.2: Example of a `Solution` schema generated by an agent.

Trace

The `Trace` schema is an immutable record of a benchmark run. It links a `Solution` to a `Definition` for specific input shapes and includes runtime statistics, correctness outcomes, and environment metadata (device type, GPU model, etc.). Traces provide the foundation for regression tracking, reproducibility, and best-solution selection.

```

{
  "definition": "matmul_bf16",
  "solution": "agent_llama3_triton_v1",
  "shapes": {"M": 4096, "N": 4096, "K": 4096},
  "runtime_ms": 3.21,
  "correctness": true,
  "device": "NVIDIA B200",
  "timestamp": "2025-06-15T14:32:00Z"
}

```

Figure 5.3: Example of a Trace schema recording a benchmark result.

Together, these three schemas (Definition, Solution, Trace) form the backbone of FlashInfer Bench. They provide structure, transparency, and a common language for benchmarking across diverse workloads and implementations.

We generated benchmark data in the form of the above schemas by tracing live model inference on NVIDIA B200 GPUs using SGLang. Kernels were collected from:

- LLaMA 3.1 8B
- DeepSeek-V2-Lite
- DeepSeek 3

These runs produced structured Definition, Solution, and Trace files that fed into FlashInfer Bench.

We were able to generate this benchmark data by using an editable FlashInfer package. We extended FlashInfer with a customized logging-enabled build to capture every kernel dispatched at runtime and chosen by the kernel selector. This integration extracted input/output tensor shapes, kernel entry point, and device and performance metadata.

Lastly, we created some user interfaces to ease the usage of the benchmark.

We designed and implemented the *Inspector*, a debugging and visualization tool that provides compilation and runtime logs, correctness failure reasons, and latency breakdowns by operator. The Inspector also enables users to explore benchmark history, input shapes, and performance deltas across time.

Finally, we built a public-facing leaderboard interface showcasing the best-performing kernels per workload. Each entry links directly to its Definition, Solution, and Trace. The Leaderboard supports per-workload rankings sorted by speedup and correctness, transparent provenance for benchmarking results, and a foundation for an open submission hub for agents and developers.

5.3 How this Sets the Stage for Future Agent Integration

The design of FlashInfer Bench establishes a foundation that directly supports the integration of agentic systems for kernel generation, refinement, and evaluation. By providing a standardized schema (Definition, Solution, Trace), a reproducible evaluation pipeline, and persis-

tent benchmark artifacts, the benchmark resolves many of the limitations encountered in earlier experimental efforts with LLM kernel agents.

Agents must be able to generate, test, and refine kernels in an iterative loop. For this to be meaningful, each iteration must produce results that can be compared across time, hardware, and even different agents. FlashInfer Bench achieves this by:

- Enforcing structured input/output definitions for every workload.
- Persistently recording metadata, performance results, and kernel provenance in immutable `Trace` objects.
- Ensuring that identical workloads and shapes yield consistent evaluation across devices and environments.

This enables agents to reliably learn from their own prior attempts or compare against other submissions.

Unlike prior efforts, FlashInfer Bench does not limit evaluation to agent-generated kernels alone. Human-written and compiler-generated baselines are included in the same schema and evaluation process. This ensures that agentic systems are judged not only relative to one another, but also against state-of-the-art expert implementations. For future agents, this sets realistic performance ceilings and high-quality exemplars to emulate.

FlashInfer Bench already exposes the interfaces needed for seamless agent integration. The `Benchmark` object provides a programmable API for compiling, executing, and validating kernels. The `TraceSet` analytics layer allows agents to query historical results, track regressions, and automatically surface best-performing kernels. The CLI and Inspector tooling provide feedback channels that can be looped into an agent’s reasoning process, enabling automatic diagnosis of failures and targeted refinement.

This positions FlashInfer Bench not just as a static benchmark, but as a runtime environment where agents can continuously improve.

Finally, by combining reproducibility, shared datasets, and a public-facing leaderboard, FlashInfer Bench sets the stage for collaborative progress. Agents, human developers, and compiler authors can all submit their solutions into the same ecosystem, enabling cross-pollination of techniques between humans, compilers, and agents. In addition, it allows longitudinal tracking of how agentic systems evolve. It also provides a sustainable path toward agents that can meaningfully contribute to the performance-critical GPU kernel development pipeline.

In summary, FlashInfer Bench bridges the gap between exploratory agent workflows and standardized benchmarking practices. This foundation makes it possible to evaluate agents fairly, measure their improvements rigorously, and ultimately accelerate the integration of agentic systems into real-world GPU software stacks.

Chapter 6

Results and Discussion

6.1 Synthesis & Cross-Cutting Insights

Taken together, the three systems, the Web LLM Agent, the LLM Kernel Agent, and the FlashInfer Benchmark, trace a trajectory from exploratory agent prototypes to structured infrastructures for kernel optimization.

The Web LLM Agent established the foundations of agentic orchestration. By embedding large language models within interactive loops that called external APIs, it demonstrated the feasibility of extending LLM reasoning into software execution. The lessons learned here, particularly around state persistence, user feedback, and error recovery, shaped the design of the later systems.

The LLM Kernel Agent advanced this paradigm into the domain of performance-critical code. It reframed GPU kernel generation as a sequential decision process, where the agent iteratively refined candidate implementations in response to benchmark feedback. This introduced closed-loop optimization as a central motif. However, the lack of standardized workloads and reproducible evaluation environments limited its scalability and made comparisons across runs difficult.

The FlashInfer Benchmark addressed these gaps by grounding kernel optimization in a structured, extensible evaluation framework. Through its schema-based design of Definitions, Solutions, and Traces, it provided the reproducibility, traceability, and extensibility that earlier systems lacked. With its runtime and analytics layers, FlashInfer transformed ad-hoc kernel experiments into a systematic benchmarking ecosystem, capable of supporting both human-authored and agent-generated solutions.

In synthesis, these systems reveal a pattern of increasing structure and generality. What began as interactive prototypes matured into an extensible evaluation backbone. The progression highlights three cross-cutting insights: (1) feedback loops are indispensable for guiding both human and agent iteration, (2) structured traces and provenance are essential for reproducibility and long-term progress, and (3) standardized, evolving benchmarks form the backbone that enables agentic systems to scale from isolated experiments to community-driven optimization.

6.2 Common Architectural Motifs

Across the three systems explored in this thesis, several recurring design motifs emerge. These architectural patterns highlight both the opportunities and constraints of building agentic systems that operate in performance-sensitive, feedback-driven domains.

Each system relies on iterative refinement guided by feedback. The Web LLM Agent incorporated user and API responses as corrective signals. The LLM Kernel Agent used benchmark metrics, specifically correctness and latency, to guide iterative kernel synthesis. FlashInfer Bench formalized this by embedding benchmark results into immutable `Trace` objects, creating a persistent record of agent-environment interactions. The motif of feedback loops thus scales from informal user prompts to structured evaluation pipelines.

A second motif is the reliance on structured representations to encode tasks, outputs, and provenance. The Kernel Agent implicitly defined workloads and candidate implementations, though without formal schemas. FlashInfer Bench extended this by introducing explicit schemas for `Definition`, `Solution`, and `Trace`, ensuring reproducibility and comparability across runs. Schemas act as the bridge between flexible agentic generation and the rigid requirements of benchmarking.

All systems distinguish between execution/runtime layers and analysis/introspection layers. The Web LLM Agent separated query execution from interaction. The LLM Kernel Agent separated kernel execution from agent reasoning. FlashInfer Bench explicitly encodes this split into the `Benchmark` (execution engine) and `TraceSet` (analytics layer). This separation enables extensibility: runtime engines can evolve independently of how results are aggregated, visualized, or interpreted.

Finally, the motif of logging underpins each successive system. While early iterations lacked persistent logging, later systems recognized that transparent, traceable records of actions, inputs, and outputs are indispensable for scientific reproducibility and community collaboration. FlashInfer Bench operationalizes this by making logging a first-class citizen in its architecture.

In sum, the motifs of feedback loops, schemas, execution-analysis separation, and logging represent a shared architectural grammar. They both explain the continuity across projects and illuminate design principles that will generalize to future agentic systems.

6.3 Recurring Bottlenecks & Pivot Points

In tracing the trajectory across the Web LLM Agent, LLM Kernel Agent, and FlashInfer Bench, certain challenges repeatedly surfaced. These bottlenecks highlight not only the technical limits of current systems but also the strategic pivot points that shaped subsequent designs.

In every system, evaluation speed and fidelity emerged as a bottleneck. For the Web LLM Agent, responsiveness was constrained by model inference and external API calls. For the LLM Kernel Agent, correctness and performance testing (compilation, execution, and profiling) proved slower than synthesis, throttling iteration speed. FlashInfer Bench addressed this by treating evaluation as a first-class component (`Trace`), but even here the throughput of benchmark runs limits the pace of exploration. This recurrent bottleneck shows that the capacity to evaluate constrains the capacity to innovate.

Agents often struggled when tasks lacked clear specification. In the Web LLM Agent, vague prompts led to brittle responses. In the LLM Kernel Agent, underspecified workloads caused mismatches between generated kernels and evaluation inputs. FlashInfer Bench resolved this by formalizing task schemas (`Definition`), making specifications unambiguous and enforceable. This pivot from implicit to explicit specification represents a turning point in agent design.

Another recurring pain point was the opacity of agent failures. Users of the Web LLM Assistant often lacked insight into why an answer failed. The LLM Kernel Agent users faced cryptic compilation or runtime errors without actionable traces. FlashInfer Bench introduced the Inspector to surface logs, failure reasons, and performance breakdowns, directly addressing this bottleneck. The pivot toward transparent logging and human-readable diagnostics was critical for moving from research prototypes to usable systems.

Finally, each project wrestled with the tradeoff between allowing agents freedom to explore versus constraining them with structure. The Web LLM Assistant leaned toward flexibility but often produced off-target results. The LLM Kernel Agent experimented with bounded search but struggled with rigid evaluation. FlashInfer Bench found a balance by embedding structure (schemas, traces) while still permitting arbitrary implementation languages and styles. This repeated tension forced a pivot toward hybrid approaches: structured enough for comparability, flexible enough for creativity.

Together, these recurring bottlenecks and pivots chart the trajectory from loosely structured, fragile systems to rigorously specified, transparent, and evaluable agentic infrastructures. They represent both hard-won lessons and design invariants for future work.

Chapter 7

Conclusion

This thesis traced the development of systems that progressively bring agents closer to the problem of GPU kernel optimization. Across three distinct but connected projects, the Web LLM Assistant, the LLM Kernel Agent, and FlashInfer Bench, we established both technical foundations and design principles for building agentic systems in high-performance computing.

The contributions span multiple layers of this space. The Web LLM Agent demonstrated how lightweight agent architectures can be embedded into developer-facing tools, offering responsive though imperfect feedback loops. The Kernel Agent extended this paradigm into the GPU optimization domain, where correctness, performance, and iteration speed are tightly coupled, and in doing so revealed bottlenecks of evaluation and the importance of structured specification. FlashInfer Bench then operationalized these lessons into a full-stack benchmarking and analytics platform. With schemas for workload definition, solution representation, and trace logging, it created a reusable infrastructure for evaluating kernels across human-written, compiler-generated, and LLM-generated implementations. Alongside these systems, tooling such as the TraceSet API, the command-line interface, and the Inspector provided transparent evaluation, human-readable diagnostics, and intuitive entry points that help close the feedback loop for both developers and agents. Taken together, these systems offered not just practical tools but also design insights: evaluation speed constrains innovation, explicit schemas reduce ambiguity, and transparency accelerates iteration.

The arc of this work points toward a future in which GPU kernel optimization is no longer dominated by manual, expert-driven tuning but is continuously refined by agentic systems. A mature realization of this vision would involve agents that can propose, evaluate, and iteratively refine kernels within self-contained feedback cycles. These systems will likely not be monolithic, but collaborative, with multiple LLMs and specialized agents coordinating across programming languages, hardware backends, and workloads. In such a setting, benchmarking infrastructure must also evolve, enabling not only static evaluation but also the dynamic assessment of adaptive behaviors. Human oversight will remain essential, however, and interfaces like the Inspector illustrate how developers might guide, constrain, and debug agentic exploration rather than merely consume its outputs. In this way, performance engineering becomes iterative, data-driven, and collaborative between humans and agents.

At the same time, several broader challenges and open questions remain. Reliability and trust are central: how do we ensure correctness and safety when deploying agent-generated kernels

in production systems? Evaluation remains the rate-limiting step, raising the question of how to scale benchmarking infrastructures without prohibitive costs. There is also the challenge of integration and how agentic kernel optimizers will coexist with established compiler stacks, libraries, and heterogeneous hardware ecosystems. Finally, the human role must be reconsidered: as agents take on more of the optimization workload, what aspects of performance engineering remain uniquely human, and how do we design systems that amplify rather than replace expertise?

In summary, this thesis demonstrates that building the scaffolding for agentic GPU-kernel optimization is both technically feasible and conceptually urgent. By aligning agent architectures, benchmarking infrastructures, and transparent feedback mechanisms, we move closer to a future where performance engineering is democratized, accelerated, and continuously evolving through human-agent collaboration.

Bibliography

- [1] J Bai et al. Qwen-vl: A frontier large vision-language model with versatile abilities. *arXiv preprint arXiv:2308.12966*, 2023. 2.1
- [2] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33:1877–1901, 2020. 1.2
- [3] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. In *Proceedings of Deep Learning Systems*, 2014. 1.1
- [4] J Li et al. Webvoyager: Building and evaluating llm agents for web navigation. In *NeurIPS 2023*, 2023. 2.1
- [5] NVIDIA. Cutlass: Cuda templates for linear algebra subroutines. <https://github.com/NVIDIA/cutlass>, 2023. 1.1
- [6] NVIDIA. Tensorrt: Nvidia deep learning inference optimizer and runtime. <https://developer.nvidia.com/tensorrt>, 2023. 1.1
- [7] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022. 1.2
- [8] Timo Schick, Aakanksha Dwivedi-Yu, Roberto Dessì, et al. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023. 1.1
- [9] Philippe Tillet, H Samuel Kung, and William Cox. Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19, 2019. 1.1
- [10] Y Wang et al. Ui-tars: Benchmarking large language model agents on ui navigation tasks. *arXiv preprint arXiv:2309.11295*, 2023. 2.1
- [11] H Zhang et al. Kernelbench: Benchmarking llm agents for gpu kernel generation. *arXiv preprint arXiv:2310.12936*, 2023. 2.2, 4.1
- [12] J Zhang et al. browser-use: Evaluating llm agents on web tasks. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2023. 2.1