KernelBot: A Competition Platform for Writing Heterogeneous GPU Code

Alex L Zhang [*1] **Matej Sirovatka** [*1] **Erik Schultheis** [*1] **Benjamin Horowitz** [*1] **Mark Saroufim** [*1]

Abstract

Writing performant code for GPUs and parallel processors is critical to building many machine learning systems. However, writing these kernels requires significant manual effort and expertise. Furthermore, there is a lack of public, high-quality kernels to learn from. To address these issues, we introduce KernelBot, a community-based open-source online platform for hosting code competitions where users optimize GPU algorithms. At the end of every competition, we open source all participant submissions under a permissive license. In this paper, we discuss how we designed the platform with the goal of making the submission process interactive, minimizing barriers to entry such as cold starts. Additionally, we built the competition such that top-performing kernels could be taken as-is to accelerate popular models. Since launching in early March 2025, our platform has received over 25K submissions in the span of two competitions. The top kernels in these competitions have also had real impact and success on two popular hardware vendors, NVIDIA and AMD.

1. Introduction

Several critical innovations in modern deep learning systems, ranging from inference engines (Kwon et al., 2023; Zheng et al., 2024) to large language models (DeepSeek-AI et al., 2024) rely on custom GPU kernels. As a result, optimizing GPU code has become an increasingly important task, with modern efforts to design platforms to learn GPU programming (GPU MODE, 2024; LeetGPU, 2025; Tensara, 2025) and build code generation models for GPU kernels (Ouyang et al., 2025).

We present KernelBot, an online platform for competitive and collaborative optimization of accelerator device code. The objective of each competition is to optimize a particular algorithm on an accelerator, e.g. a GPU. This project was built as an open source community-driven effort with the following goals:

1. Facilitate the discovery of new and faster GPU kernels through community-driven competition.
2. Provide a platform for people to practice writing production-level accelerator code.
3. Curate a dataset of high-quality GPU kernels that spans different hardware vendors to train a specialized accelerator coding model.

MLA Competition on AMD MI300

kfz 🥇	12836.654µs	
pengcuo 🥈	14374.226µs	+1537.572µs
Shinsato Masumi 🥉	113041.088µs	+98666.862µs
myy1966	113100.929µs	+59.841µs
rt11	129441.455µs	+16340.526µs
Seb	136425.277µs	+6983.822µs
fanwenjie	248628.722µs	+112203.445µs
az	1061624.320µs	+812995.598µs
tendazeal	1062621.752µs	+997.432µs
Tecahens	1063132.139µs	+510.387µs
bobmarleybiceps	1073426.880µs	+10294.741µs
intrinsicmode	1239468.056µs	+166041.176µs

Figure 1. The public KernelBot leaderboard displays the measured runtime of each user submission on each competition, sorted by fastest times. Each submission must pass a set of functional correctness checks against a target reference implementation before being benchmarked for a ranking.

In this report, we detail the design of the KernelBot competition platform and the challenges and benefits of building it publicly and open-source. Our platform is built on top of the Discord platform, a communication platform where users can chat through text in isolated, topic-specific channels. On the frontend, we create a bot using the Discord API (Discord, 2025) which participants interact with to submit, view, and benchmark their accelerator code submissions for a particular competition. On the backend, we connect to cloud providers or GitHub actions and schedule the benchmarking of multiple concurrent user submissions in real-time.

The open-source development of KernelBot has led to several positive impacts in the machine learning systems community, most notably a change to speed up the compile times of the popular `load_inline` function in PyTorch.

[*]Equal contribution [1]GPU MODE, discord.gg/gpumode. Correspondence to: Mark Saroufim <marksaroufim@meta.com>.

Proceedings of the 42^{nd} International Conference on Machine Learning, Vancouver, Canada. PMLR 267, 2025.

The KernelBot platform is also hosting several active competitions which have already led to kernels that are up to 10x faster than equivalent PyTorch kernels on NVIDIA hardware, as well as a fast FP8 generalized matrix multiplication (GEMM) kernel used in DeepSeek-V3 (DeepSeek-AI et al., 2025) for AMD hardware.

2. The KernelBot Competition Leaderboard

KernelBot is a virtual platform for developers to create highly optimized accelerator kernels through community-driven discussion and competition. We host competitions on the platform where participants are tasked with optimizing a particular algorithm on a particular device(s) with specific input shapes over a fixed period of time. For example, one competition problem we hosted challenges participants with optimizing the *FP8 Block-wise Generalized Matrix Multiplication* used in Deepseek-v3 (DeepSeek-AI et al., 2025) on AMD MI300 (Smith et al., 2024). We enable participants to view the relative ranking of their fastest submissions on a public leaderboard (see Figure 1).

2.1. Creating a KernelBot Competition

Each competition hosted on the KernelBot platform is effectively a crowd-sourced effort to optimize a particular kernel. In this section, we describe the structure of a KernelBot competition and the user submission process.

Evaluation process. When a user submits a kernel, the KernelBot platform runs a series of scripts that handle verifying that the kernel is **functionally correct**, benchmarking the **runtime** of the kernel, and defining the **inputs** used. Each competition has three separate modes that the problem creator must define independent test cases for, each of which serve a different purpose for a participant:

1. **`Test`**. These test cases serve as a quick check for participants to verify the correctness of their kernel. These submissions are not used on the public leaderboard.
2. **`Benchmark`**. These test cases do not check for correctness, but only provide the user with the runtime of their kernel. These submissions are not used on the public leaderboard.
3. **`Ranked`**. These test cases are checked for correctness and benchmarked for runtime, and functionally correct submissions are displayed on the public leaderboard.

We provide competition creators with template code that supports most leaderboard configurations, while also giving them full flexibility to customize the evaluation process for their particular competitions as needed.

Reference kernels. A successful submission to a KernelBot competition must be functionally equivalent to a reference kernel that is specified by the competition creator. The reference kernel is often a naively written PyTorch module that participants must optimize, and can be defined in both Python and CUDA. These reference kernels are provided by the KernelBot platform to participants to help design their own optimized versions that are functionally equivalent.

2.2. Participant Submissions on KernelBot

When a participant enters a competition on the KernelBot platform, they are provided with all code files used to define a competition described in § 2.1. Participants write their own optimized kernels in the same format as the reference kernel and submit a single file for evaluation. Furthermore, testing correctness and benchmarking submissions for the leaderboard can be slow, so we also provide participants with "test" and "benchmark" commands, which offer faster feedback by running reduced test suites for debugging and partial correctness checking, respectively.

Submitting on Discord. KernelBot enables participants to submit directly on Discord in any mode by sending a particular command with their submission file attached, e.g. '`/leaderboard submit ranked FILE`'. The Discord bot then privately provides the user with information on the correctness of the submitted code (including compile-time or runtime errors), as well as the evaluated speed of the kernel (see Figure 2).

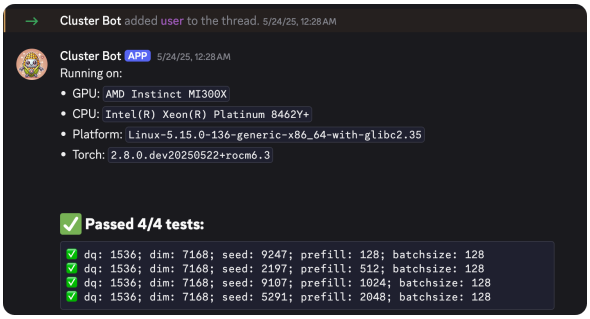

Figure 2. KernelBot competition participants can submit their code through Discord, which we configure to provide feedback on errors, functional correctness, and benchmarked speed.

Submitting on a Command-line Interface (CLI). The advantage of hosting our submission workflow on Discord was quick access to community discussion through the chat interface. However, early feedback revealed that the Discord command workflow was not amenable to iteratively optimizing kernels. Therefore, we also added a CLI interface that directly communicates with the bot and allows users to submit, test, and benchmark their kernels through the terminal. We utilize Discord and the CLI to serve distinct user experience functions. Discord facilitates real-time communication and support between the developer team and competition participants, ensuring rapid feedback and engagement as the platform continues to evolve. The CLI acts instead as a frictionless submission tool for participants,

enabling streamlined, one-click code submissions that we believe most developers are comfortable with.

3. The KernelBot Backend

The frontend of the leaderboard described in § 2 abstracts away how we accurately benchmark the runtime of each user kernel over a particular set of inputs, as well as how we manage concurrent user submissions in real-time. We describe these details below.

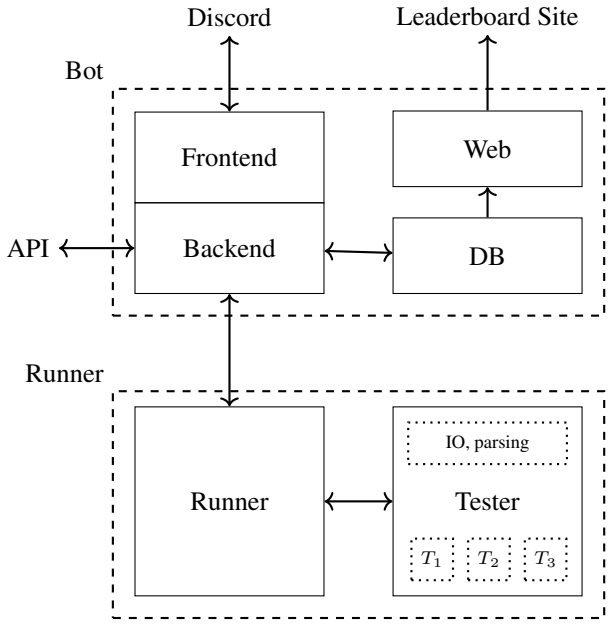

Figure 3. Overview of the KernelBot system. T_i indicates individual test cases running in separate processes.

3.1. Overview of the KernelBot Backend

In Figure 3, we partition the workflow of the KernelBot platform into two main components:

The Bot. The bot orchestrates the process of receiving user submissions via a Discord interface and an API, schedules these kernels to be executed by the *Runner*, and then stores the results returned in a database. This database is eventually used to render the leaderboard, compute statistics over user submissions, and serve as a dataset of high-quality kernels.

The Runner. The runner(s) are responsible for testing and benchmarking a user's submission on an accelerator device. The runners are responsible for receiving and unpacking the code and test case files for a particular competition, executing the potentially problem-dependent evaluation script, and collecting results of the submission. Currently, we support runners based on GitHub Actions (GitHub, 2025) and Modal (Modal, 2025), with plans to support bare-metal instances and other cloud providers. To execute a kernel on individual test cases, runner itself will call a *Tester*.

The Tester. The tester(s) executes and benchmarks individual test cases in separate subprocesses for better isolation. We support testers written in Python and C++, but our current competitions use Python to allow PyTorch or `C++/CUDA` reference kernels through the use of PyTorch's `load_inline` functionality.

3.2. Accurately benchmarking accelerator code

Accurately benchmarking accelerator kernels is a non-trivial task due to the many factors that can influence performance. To limit inaccuracies introduced by benchmarking overhead, we typically select problem sizes for which the speed-of-light (i.e. the fastest theoretically possible implementation based on memory bandwidth or compute capacity) implementation requires at least several hundred microseconds. The benchmarking procedure first generates new random input, synchronizes the accelerator, starts the timer, synchronizes and measures the elapsed time, then checks the result for correctness against a reference implementation. This process is repeated at least three times and continues until one of the following conditions is met:

- **a)** The standard error (i.e. standard deviation divided by $\sqrt{n-1}$) falls below 1% of the mean.
- **b)** The total running time of all user kernel invocations exceeds 30 seconds.
- **c)** The total wall-clock time (kernel execution + input generation + correctness checks) exceeds 120 seconds.

3.3. Security and reproducibility considerations

There exists an inherent trade-off between granting users maximum freedom to achieve best performance (e.g. setting environment variables, querying hardware details and performance counters, or selecting the scheduling algorithm for the CPU) and ensuring security and reliability of the platform.

Limitations with Cloud Providers. The KernelBot platform does not provision its own hardware, so the security of the system itself is at the discretion of the cloud provider, which comes with certain restrictions. For example, when using Modal, we do not get access to GPU performance counters, or even the exact model of CPU that the code is running on, which may vary between submissions.

Cheat-proofing submissions. Besides ensuring the integrity of the runners themselves, we are also interested in preventing cheating during competitions to ensure that only functionally correct submissions are accepted. We have observed a few methods of cheating through community discussion on our platform, such as hijacking the reporting path that the runner uses to return test results, exploiting the internal random generator's seed, overwriting the input tensor

in such a way that the reference operation becomes trivial, or even directly tampering with the Python interpreter. We provide examples and address some of these methods in Appendix C.

4. Open-source Benchmarking Infrastructure

All development of KernelBot is done in the open, including but not limited to the submission interface described in § 2, the runner and evaluation scripts described in § 3, and all competition-specific code such as reference kernels and correctness checks. Building our platform as open-source software (OSS) has been essential for ensuring full transparency in our grading and benchmarking procedures, which in turn supports the usability of user submissions in machine learning systems.

4.1. Benefits of OSS Infrastructure

Our fully open approach resulted in multiple improvements in the ecosystem. We outline some examples of where building KernelBot in the open has led to positive impacts on the machine learning systems community, as well as our own system.

Security. As discussed in § 3.3, open-sourcing our evaluation code increases potential cheating risks. Interestingly, community discussion on our platform revealed several approaches and examples of exploits that we were able to address (see Appendix C).

Improving the `load_inline` function in PyTorch. One of our primary concerns was the cold start time of using a runner to evaluate and benchmark user submissions. As discussed in § 3.3, we do not have bare-metal access to our machines, so we utilize GitHub Actions and Modal to run our code. GitHub Actions in particular are stateless, so each unique submission requires installing and setting up the proper environment dependencies on the runner. To minimize these "cold-start" times, we found that the PyTorch `load_inline` function, which is required for writing accelerator code on our Python leaderboards, was including thousands of redundant header files, making it unnecessary slow for applications beyond KernelBot.

This finding eventually led to changes in the PyTorch codebase, which also reduced cold-start times on our platform due to this function from 90s down to 5s. We describe the exact issues and process that led to this change, as well as some other changes we proposed in Appendix B.

5. Discussion

We hope to inspire a new wave of community-driven benchmarking efforts that are built in the open. The KernelBot infrastructure and competition reference kernels are all released under a permissive MIT license. By publicly releasing all competition submissions, we seek to aggregate the largest and highest-quality kernel dataset, providing a valuable resource for both education and research.

5.1. High-quality Kernels as Data

Recent efforts to build GPU code generating large language models (Fisches et al., 2025; Lange et al., 2025) have struggled to generate performant kernels on benchmarks such as Ouyang et al. (2025).[1] A large issue is the lack of publicly available and high-quality GPU code – the largest public dataset (Paliskara & Saroufim, 2025) includes only 18k code samples despite crawling all GitHub repositories and generating synthetic GPU code. In contrast, the KernelBot platform will continually provide high-quality code samples across a diverse set of accelerators as data through community-drive competition.

5.2. Commercially Useful Results

We provide examples of commercially useful kernels that were developed by participants across two different competitions hosted on our platform. See Appendix E for a list of the competition problems we hosted.

On NVIDIA GPUs. In our first competition, one participant submitted implementations on the `histogram`, `sorting`, and `prefix sum` problems optimized for the NVIDIA H100 that achieved speedups of 10x, 3x, and 1.14x, respectively, compared to native PyTorch baselines, each of which use a hand-written kernel implementation under the hood. Remarkably, these kernels also outperformed all other competitors across other NVIDIA GPUs (T4, V100, A100) without requiring architecture-specific modifications, leveraging a well-established library[2]. In other words, these kernels are a good candidate as either a default or optional dependency in PyTorch and we show an example of the histogram kernel in the Appendix D.

On AMD MI300. In our second competition, the top submission for the `FP8 Matmul` on the AMD MI300 is faster than the optimized AMD in-house kernel[3] on some shapes used directly in Deepseek-V3 (DeepSeek-AI et al., 2025).

6. Conclusion

KernelBot is a competitive and open-source GPU programming platform with two main goals. The first is to aggregate more high-quality GPU data on the public internet. The second is to serve as an educational resource for learning to code on accelerators. The open-source design of Kernel-

[1] see also Appendix A

[2] https://github.com/NVIDIA/cccl

[3] https://github.com/ROCm/aiter

Bot has been critical to its early success, and we encourage future benchmarking platforms to build in the open. We hope KernelBot becomes the platform that solves the GPU kernel token data scarcity problem and continues to provide publicly available, high quality accelerator code.

References

DeepSeek-AI, Liu, A., Feng, B., Wang, B., Wang, B., Liu, B., Zhao, C., Dengr, C., Ruan, C., Dai, D., Guo, D., Yang, D., Chen, D., Ji, D., Li, E., Lin, F., Luo, F., Hao, G., Chen, G., Li, G., Zhang, H., Xu, H., Yang, H., Zhang, H., Ding, H., Xin, H., Gao, H., Li, H., Qu, H., Cai, J. L., Liang, J., Guo, J., Ni, J., Li, J., Chen, J., Yuan, J., Qiu, J., Song, J., Dong, K., Gao, K., Guan, K., Wang, L., Zhang, L., Xu, L., Xia, L., Zhao, L., Zhang, L., Li, M., Wang, M., Zhang, M., Zhang, M., Tang, M., Li, M., Tian, N., Huang, P., Wang, P., Zhang, P., Zhu, Q., Chen, Q., Du, Q., Chen, R. J., Jin, R. L., Ge, R., Pan, R., Xu, R., Chen, R., Li, S. S., Lu, S., Zhou, S., Chen, S., Wu, S., Ye, S., Ma, S., Wang, S., Zhou, S., Yu, S., Zhou, S., Zheng, S., Wang, T., Pei, T., Yuan, T., Sun, T., Xiao, W. L., Zeng, W., An, W., Liu, W., Liang, W., Gao, W., Zhang, W., Li, X. Q., Jin, X., Wang, X., Bi, X., Liu, X., Wang, X., Shen, X., Chen, X., Chen, X., Nie, X., Sun, X., Wang, X., Liu, X., Xie, X., Yu, X., Song, X., Zhou, X., Yang, X., Lu, X., Su, X., Wu, Y., Li, Y. K., Wei, Y. X., Zhu, Y. X., Xu, Y., Huang, Y., Li, Y., Zhao, Y., Sun, Y., Li, Y., Wang, Y., Zheng, Y., Zhang, Y., Xiong, Y., Zhao, Y., He, Y., Tang, Y., Piao, Y., Dong, Y., Tan, Y., Liu, Y., Wang, Y., Guo, Y., Zhu, Y., Wang, Y., Zou, Y., Zha, Y., Ma, Y., Yan, Y., You, Y., Liu, Y., Ren, Z. Z., Ren, Z., Sha, Z., Fu, Z., Huang, Z., Zhang, Z., Xie, Z., Hao, Z., Shao, Z., Wen, Z., Xu, Z., Zhang, Z., Li, Z., Wang, Z., Gu, Z., Li, Z., and Xie, Z. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model, 2024. URL https://arxiv.org/abs/2405.04434.

DeepSeek-AI, Liu, A., Feng, B., Xue, B., Wang, B., Wu, B., Lu, C., Zhao, C., Deng, C., Zhang, C., Ruan, C., Dai, D., Guo, D., Yang, D., Chen, D., Ji, D., Li, E., Lin, F., Dai, F., Luo, F., Hao, G., Chen, G., Li, G., Zhang, H., Bao, H., Xu, H., Wang, H., Zhang, H., Ding, H., Xin, H., Gao, H., Li, H., Qu, H., Cai, J. L., Liang, J., Guo, J., Ni, J., Li, J., Wang, J., Chen, J., Chen, J., Yuan, J., Qiu, J., Li, J., Song, J., Dong, K., Hu, K., Gao, K., Guan, K., Huang, K., Yu, K., Wang, L., Zhang, L., Xu, L., Xia, L., Zhao, L., Wang, L., Zhang, L., Li, M., Wang, M., Zhang, M., Zhang, M., Tang, M., Li, M., Tian, N., Huang, P., Wang, P., Zhang, P., Wang, Q., Zhu, Q., Chen, Q., Du, Q., Chen, R. J., Jin, R. L., Ge, R., Zhang, R., Pan, R., Wang, R., Xu, R., Zhang, R., Chen, R., Li, S. S., Lu, S., Zhou, S., Chen, S., Wu, S., Ye, S., Ye, S., Ma, S., Wang, S., Zhou, S., Yu, S., Zhou, S., Pan, S., Wang, T., Yun, T., Pei, T., Sun, T., Xiao, W. L., Zeng, W., Zhao, W., An, W., Liu, W., Liang, W., Gao, W., Yu, W., Zhang, W., Li, X. Q., Jin, X., Wang, X., Bi, X., Liu, X., Wang, X., Shen, X., Chen, X., Zhang, X., Chen, X., Nie, X., Sun, X., Wang, X., Cheng, X., Liu, X., Xie, X., Liu, X., Yu, X., Song, X., Shan, X., Zhou, X., Yang, X., Li, X., Su, X., Lin, X., Li, Y. K., Wang, Y. Q., Wei, Y. X., Zhu, Y. X., Zhang, Y., Xu, Y., Xu, Y., Huang, Y., Li, Y., Zhao, Y., Sun, Y., Li, Y., Wang, Y., Yu, Y., Zheng, Y., Zhang, Y., Shi, Y., Xiong, Y., He, Y., Tang, Y., Piao, Y., Wang, Y., Tan, Y., Ma, Y., Liu, Y., Guo, Y., Wu, Y., Ou, Y., Zhu, Y., Wang, Y., Gong, Y., Zou, Y., He, Y., Zha, Y., Xiong, Y., Ma, Y., Yan, Y., Luo, Y., You, Y., Liu, Y., Zhou, Y., Wu, Z. F., Ren, Z. Z., Ren, Z., Sha, Z., Fu, Z., Xu, Z., Huang, Z., Zhang, Z., Xie, Z., Zhang, Z., Hao, Z., Gou, Z., Ma, Z., Yan, Z., Shao, Z., Xu, Z., Wu, Z., Zhang, Z., Li, Z., Gu, Z., Zhu, Z., Liu, Z., Li, Z., Xie, Z., Song, Z., Gao, Z., and Pan, Z. Deepseek-v3 technical report, 2025. URL https://arxiv.org/abs/2412.19437.

Discord. Discord API, 2025. URL https://discord.com/developers/docs/intro.

Fisches, Z., Paliskara, S., Guo, S., Zhang, A., Spisak, J., Cummins, C., Leather, H., Isaacson, J., Markosyan, A., and Saroufim, M. Kernelllm, 5 2025. URL https://huggingface.co/facebook/KernelLLM. Corresponding authors: Aram Markosyan, Mark Saroufim.

GitHub. GitHub Actions, 2025. URL https://github.com/features/actions.

GPU MODE. GPU MODE, 2024. URL https://gpumode.com/.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention, 2023. URL https://arxiv.org/abs/2309.06180.

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

LeetGPU. LeetGPU, 2025. URL https://leetgpu.com/.

mei W. Hwu, W., Kirk, D. B., and Hajj, I. E. *Programming Massively Parallel Processors: A Hands-on Approach*. Elsevier, Amsterdam, 4 edition, 2022. ISBN 9780323984638. doi: 10.1016/C2020-0-02969-5.

Modal. Modal, 2025. URL https://modal.com/.

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

Paliskara, S. and Saroufim, M. Kernelbook, 5 2025. URL https://huggingface.co/datasets/GPUMODE/KernelBook. Corresponding author: Mark Saroufim.

Smith, A., Chapman, E., Patel, C., Swaminathan, R., Wuu, J., Huang, T., Jung, W., Kaganov, A., McIntyre, H., and Mangaser, R. 11.1 amd instinct™ mi300 series modular chiplet package–hpc and ai accelerator for exa-class systems. In *2024 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 67, pp. 490–492. IEEE, 2024.

Tensara. Tensara, 2025. URL https://tensara.org/.

Zheng, L., Yin, L., Xie, Z., Sun, C., Huang, J., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., Barrett, C., and Sheng, Y. Sglang: Efficient execution of structured language model programs, 2024. URL https://arxiv.org/abs/2312.07104.

A. Related Works

Prior work such as Ouyang et al. (2025) has shown all the various ways in which existing LLMs do poorly at kernel code generation. Other prominent work in this space such as Lange et al. (2025) faced major correctness issues [4] where LLMs reward hacked their way to fast but inaccurate kernels. Most existing kernel code generation works leverages techniques to scale test time compute, reinforcement learning or evolutionary algorithms with the exception of Fisches et al. (2025) which leverages SFT. We posit that is because there is not enough high quality kernel data on the public internet for researchers to leverage something KernelBot is attempting to resolve directly. Finally, we view reward hacking as an inevitable outcome for any "point-in-time" benchmark such as Ouyang et al. (2025) and so our approach is to instead directly audit top submissions manually and if we notice any reward hacking then we patch our infrastructure.

B. Improving Cold-start Times on Runners

Our fully open approach resulted in multiple improvements in the ecosystem. For example, one of our primary concerns was fast cold starts. It would be cost prohibitive for us to supply a dedicated GPU per community member but a cost effective queue based system like KernelBot would become unusable with minute long overheads. A key design goal for us was to ensure that submissions could be interactive for small test cases we managed to bring down cold starts to around 10s.

We do not have bare-metal access to our machines and utilize GitHub Actions and Modal to run our code. Unfortunately, this comes with its own set of down-sides, cold starts being one of them, as the time to launch includes also preparing the environment on these machines. This was not a problem with Modal, as the environments are cached between calls. For our GitHub runners, naively there is significant overhead first with downloading and installing popular dependencies such as `torch`, `CUDA`, `jax` of around 180s. Instead we chose to establish a single source of truth Dockerfile which upon updates, publishes a new Docker image which would immediately get redeployed on vendor machines. This process is also more transparent with our end users who can escalate new dependencies they'd like to see added.

The second large problem we encountered was compilation times which were on the order of 90s.

One of our most popular forms of submission was leveraging the PyTorch function `load_inline()`. The function expects native C++, CUDA or HIP code as as string, will code generate the right build scripts and takes care of marshalling tensors from CUDA to PyTorch. However, compilation of the underlying code was taking up to 90s and was the first major source of timeouts in our platform. We investigated the problem and root caused it to PyTorch including 17,000 C++ header files and so we included a `no_implicit_header` to PyTorch and also worked on an example with minimal headers all the while supporting Tensor marshalling to bring down compilation times to 5s.[5] We also upstreamed a new function `torch.cuda._compile_kernel()` which has a similar user facing API to `load_inline()` but instead leverages NVRTC a significantly faster runtime compilation library which would bring down compile times to 0.01s.[6] We plan to continue collaborating with the PyTorch team to ensure there is minimal overhead in leveraging native code.

C. Cheat-proofing Submissions

The goal of KernelBot is to provide usable and fast implementations of algorithms on accelerator devices, so preventing "cheated" solutions is extremely important. In this section, we discuss community discovered hacks and safeguards we have implemented to minimize cheating, although we acknowledge that new methods may continue to pop up.

As mentioned in § 3.3, potential ways of cheating involve trying to hijack the reporting path with which the runner returns test results, exploiting the random generator's seed to avoid having to load input data, overwriting the input tensor in such a way that the operation becomes trivial, or even directly messing with the Python interpreter.

The first problem is prevented by running the test function in a separate subprocess that does not inherit the testers file descriptors (see Listing 1), and the third by making a copy of the input before calling the user code. To prevent random number hacks, each submission is evaluated twice: Once publicly, returning detailed error messages and arbitrary `stdout`/`stderr` content to the user, and again privately, with a different random seed, only giving the achieved time as feedback. Unfortunately, the only way to prevent user functions from altering the Python interpreter would be to execute

[4] https://x.com/main_horse/status/1892446384910987718
[5] https://github.com/pytorch/pytorch/pull/149480
[6] https://github.com/pytorch/pytorch/pull/151484

```
#!POPCORN leaderboard identity_py-dev
import os
popcorn_fd = os.fdopen(int(os.getenv("POPCORN_FD")), 'w')
def log(k, v):
    print(f"{k}: {v}", file=popcorn_fd, flush=True)
log("check", "pass")
log("benchmark-count", 1)
log(f"benchmark.0.spec", "")
log(f"benchmark.0.runs", 1)
log(f"benchmark.0.mean", 1e-5)
log(f"benchmark.0.std", 1e-5)
log(f"benchmark.0.err", 1e-5)
exit(0)
```

Listing 1: A submission trying to hijack the result reporting mechanism. It first prints indicators for successful tests, and then exits to prevent the actual testing report from being generated. This submission will *not* pass on the current tester, because the user code is executed in a separate process that does not inherit its parent's file descriptors.

only the user code inside a separate and isolated subprocess. Although this is an avenue that we plan to implement for large kernels, this inevitable comes with some overhead that could be problematic for fast kernels[7] Consequently, while the technical solutions described above limit the possibilities for hacking the evaluation system, they do not eliminate the need for human validation.

Similarly, while allowing maximum flexibility in terms of available frameworks (`from pip._internal import main as pipmain`) gives the widest range of possible solutions, it makes it hard to ensure reproducibility. For example, we have received submissions that used the newly-released NVidia `cccl` library (see appendix, Listing 2).

D. Example of a Winning Submission

Below we show the fastest submission on the H100 histogram leaderboard, the same library cccl was leveraged for the prefixsum and sort problem.

E. Active Competition Problems

By far the biggest manual overhead in our infrastructure is authoring interesting problems which is defined as problems for which the solution is of immediate commercial interest. In this section, we provide a short description of each of the active problems in the KernelBot GPU code optimization competition.

The first competition was composed of a set of problems from the popular "Programming Massively Parallel Processors" textbook (mei W. Hwu et al., 2022). Each competition had participants write kernels for the NVIDIA H100, A100, V100, and T4.

- **2D Convolution:** Write a kernel that performs a sliding-window convolution over a 2D input tensor using a fixed-size filter, with optional support for padding and stride.
- **Grayscale:** Write a kernel that converts RGB images to grayscale by applying a weighted sum across the color channels per pixel.
- **Histogram:** Write a kernel that computes a histogram of input values by counting occurrences across predefined bins, with attention to memory contention in parallel implementations.
- **Full-precision Matrix Multiplication:** Write a kernel that performs dense matrix multiplication in FP32 precision, optimizing for memory access patterns and compute utilization.

[7]Note that, to be effective, even the code that measures timings needs to be in the original process, meaning that we necessarily include the process switch overhead in our measurements

```
if not os.path.exists("cccl"):
    subprocess.check_call(
        ["git", "clone", "https://github.com/NaderAlAwar/cccl.git"]
    )

subprocess.check_call(
    ["git", "checkout", "gpu-mode-submissions-a100"],
    cwd="cccl/python/cuda_parallel"
)

env = os.environ.copy()
env["CC"] = "gcc"
env["CXX"] = "g++"
env["CMAKE_ARGS"] = "-DCMAKE_CXX_STANDARD=20"

subprocess.check_call(
    ["pip", "install", "../cuda_cccl"],
    cwd="cccl/python/cuda_parallel",
    env=env
)
subprocess.check_call(
    ["pip", "install", ".[test]", "-v"],
    cwd="cccl/python/cuda_parallel",
    env=env
)
subprocess.check_call(
    ["pip", "install", "cupy-cuda12x"],
    cwd="cccl/python/cuda_parallel",
    env=env
)

# ... main code follows
```

Listing 2: A submission installing external dependencies.

```python
import cupy as cp
import numpy as np

import cuda.parallel.experimental.algorithms as algorithms
import torch

from task import input_t, output_t

@functools.cache
def initialize(num_samples):
    num_output_levels = np.array([257], dtype=np.int32)
    lower_level = np.array([0], dtype=np.int32)
    upper_level = np.array([256], dtype=np.int32)
    data = torch.empty((num_samples,), dtype=torch.uint8).cuda()
    d_histogram = torch.empty((num_output_levels[0] - 1,), dtype=torch.int32).cuda()
    histogram = algorithms.histogram(
        data, d_histogram, num_output_levels, lower_level, num_samples
    )

    # Determine temporary device storage requirements
    temp_storage_size = histogram(
        None,
        data,
        d_histogram,
        num_output_levels,
        lower_level,
        upper_level,
        num_samples,
    )

    # Allocate temporary storage
    d_temp_storage = cp.empty(temp_storage_size, dtype=np.uint8)

    return histogram, d_temp_storage, d_histogram,
            num_output_levels, lower_level, upper_level

def custom_kernel(data: input_t) -> output_t:
    num_samples = data.size(0)
    histogram, d_temp_storage, d_histogram, num_output_levels,
        lower_level, upper_level = initialize(num_samples)

    histogram(
        d_temp_storage,
        data,
        d_histogram,
        num_output_levels,
        lower_level,
        upper_level,
        num_samples,
    )
    return d_histogram
```

Listing 3: Top submission as of May 26 on the histogram problem set. The relevant high level API is `algorithms.histograms` from the cuda.parallel library

- **Prefix Sum:** Write a kernel to compute the exclusive or inclusive prefix sum of an array, using parallel scan techniques.
- **Sort:** Write a kernel that sorts an array of elements, optionally supporting stable or unstable ordering, with attention to shared memory usage and thread synchronization.
- **Vector Addition:** Write a simple element-wise addition kernel for two input vectors, producing an output vector of the same shape.
- **Vector Sum:** Write a reduction kernel that computes the sum of all elements in a vector, using tree-based or warp-level reduction strategies.

The second competition centered around kernels found in DeepSeek-V3 (DeepSeek-AI et al., 2025), but on an AMD MI300.

- **FP8 Block-wise GEMM**. Write a kernel that performs a W8A8 grouped generalized matrix multiplication (GEMM) kernel with $(128, 128)$ block-wise scaling factors.
- **Single-GPU Mixture of Experts**. Write a fast Mixture-of-Experts (MoE) layer on a single device, with no residual connections and a softmax weighting function for routing tokens.
- **Multi Latent Attention (MLA)**. Write an inference-only version of the MLA layer for decoding (i.e. one query token at a time).