MEGAKERNELBENCH: A SYSTEM-LEVEL BENCHMARK FOR MEGAKERNEL GENERATION AND OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Small-model, small-batch LLM decoding launches many fine-grained GPU kernels, making launch overhead a bottleneck for interactive inference. Megakernels mitigate this overhead by executing part or all of a computation graph within a single persistent work kernel. However, their performance depends jointly on operator implementation and scheduling, which existing GPU kernel benchmarks do not adequately evaluate. We present MegaKernelBench, a system-level benchmark for megakernel generation and optimization with four progressive tasks: operator generation, scheduling generation, joint operator-scheduling generation, and generation without a prebuilt runtime. It evaluates correctness, structural validity, and end-to-end performance. Existing methods can generate correct megakernels with a prebuilt runtime but generally underperform mainstream inference engines; generation without one remains more challenging. Strong operator-scheduling coupling further makes joint optimization essential. MegaKernelBench enables unified, reproducible evaluation of megakernel generation by compilers and agents.

1 INTRODUCTION

Large language model (LLM) inference is increasingly moving toward interactive applications, making low latency a critical system objective. Especially in decoding workloads with small models and batch sizes, limited computation makes kernel launch overhead a significant end-to-end performance bottleneck. Operator fusion and persistent kernels have been proposed to mitigate such overheads. More recently, megakernel has emerged as a promising direction. Instead of decomposing model execution into hundreds of individual kernels, a megakernel coordinates the execution of a subgraph, or even an entire model, within a single kernel launch. Compared with persistent kernels, megakernels extend the scope from individual operators to computation graphs comprising multiple operators. They eliminate more kernel launches and host-side scheduling overhead while creating opportunities for cross-operator data prefetching. Recent studies have explored this paradigm through compiler-based (Cheng et al., 2025; Jin et al., 2026) and agent-based (Jaber & Jaber, 2026) generation. Together, these studies suggest that megakernels are becoming an important approach to optimizing GPU inference in latency-sensitive scenarios.

A megakernel is more than a larger GPU kernel. It incorporates system-level logic for operator computation and task scheduling, thereby exhibiting characteristics of a complete inference system. Its performance depends on operator implementations, scheduling strategies, and the interactions between them. Evaluating a megakernel therefore requires examining not only end-to-end latency but also operator implementations and GPU-side scheduling behavior. This requirement distinguishes megakernel evaluation from conventional kernel evaluation. Existing kernel-generation benchmarks typically evaluate individual or fused operators while treating inter-operator scheduling as a fixed external component. In a megakernel, however, a candidate method may modify not only operator implementations but also GPU-side task partitioning, execution order, and resource allocation. Consequently, latency alone can neither attribute performance improvements to operator optimization, scheduling optimization, or their interaction nor separately assess a method's ability to generate and optimize the operator and scheduling components of a megakernel. A benchmark designed specifically for megakernel generation is thus essential infrastructure for advancing this field. Such a

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

benchmark would explain the performance advantages of megakernels, identify optimization opportunities, and guide future megakernel and compiler- and agent-based workflow design.

To fairly evaluate independent generation and optimization of the operator and scheduling components of a megakernel, their joint optimization, and whether the generated implementation is correct and meets the single-work-kernel requirement, a benchmark for megakernel generation should follow three principles: **Multi-level evaluation of generation capabilities.** The benchmark should evaluate operator and scheduling generation and optimization individually and jointly, as well as generation without a prebuilt runtime. **Multidimensional evaluation criteria.** The evaluation should consider functional correctness, compliance with the structural requirements of a megakernel, and end-to-end performance. **A unified evaluation protocol.** Candidate methods should be compared fairly and reproducibly using consistent workloads, execution environments, and evaluation criteria.

Based on these principles, we present MegaKernelBench, a system-level benchmark for evaluating the generation and optimization of high-performance megakernels. MegaKernelBench decomposes megakernel generation into operator generation and scheduling generation and defines a task hierarchy spanning single-component generation, joint generation, and generation without a prebuilt runtime. Through unified protocols for correctness, structural compliance, and performance evaluation, MegaKernelBench assesses whether a method can generate a valid and more efficient megakernel.

Experiments conducted with MegaKernelBench show that existing methods can already generate correct megakernels when provided with a prebuilt runtime, although their end-to-end latency remains higher than that of mainstream inference engines. Generation without a prebuilt runtime remains substantially more challenging. Moreover, operator implementation and scheduling are interdependent optimization spaces whose interaction determines the ultimate performance ceiling of a megakernel. These findings demonstrate that methodologies designed for individual kernels cannot adequately evaluate megakernel generation. Instead, its generation and optimization capabilities must be assessed from the perspective of a complete inference system.

This paper makes the following contributions:

(1) We introduce MegaKernelBench, the first systematic benchmark for megakernel generation and optimization, extending the scope of evaluation from individual operators to complete inference systems that incorporate device-side scheduling.

(2) We establish a hierarchy of tasks covering operator generation, scheduling generation, joint generation, and generation without a prebuilt runtime. We further provide a unified and reproducible protocol that evaluates functional correctness, structural compliance, and end-to-end performance.

(3) Using MegaKernelBench, we characterize the optimization behavior of megakernels and identify the capability boundaries of existing generation methods, providing evaluation infrastructure for future research on megakernel generation and optimization.

2 Related Work

2.1 Megakernel

Traditional LLM inference uses operators as basic execution units and performs model computation through multiple GPU kernels. To reduce kernel launch and runtime overheads in latency-sensitive scenarios, persistent kernels integrate multiple computation stages and their scheduling logic into a single kernel execution, leading to the megakernel paradigm. Hazy Research was among the first to explore megakernels for low-latency inference of Llama-1B, placing all operators of the model into a single persistent kernel to reduce launch overhead and enable cross-operator prefetching (Hazy Research, 2025). Subsequent work explored compiler-based automatic generation of megakernels. Mirage Persistent Kernel (MPK) (Cheng et al., 2025) transforms model inference into a megakernel and organizes execution using SM-level task graphs and cross-SM scheduling. Event Tensor (Jin et al., 2026) models dynamic task dependencies as dynamically instantiable event tensors to support dynamic shapes and data-dependent scheduling. More recently, AutoMegaKernel (Jaber & Jaber, 2026) has used an agent to iteratively generate and optimize megakernels.

The megakernel paradigm has also been extended to multi-GPU MoE architectures, where systems such as FlashMoE (Aimuyo et al., 2025) and MegaMoE (DeepSeek-AI, 2026a) from DeepSeek improve inference efficiency by overlapping communication and computation. Existing work has primarily focused on constructing or optimizing high-performance megakernels, but lacks systematic evaluation of operator generation, scheduling generation, and their joint optimization. In particular, compiler- and agent-driven methods still lack unified evaluation criteria for functional correctness, structural validity, and end-to-end performance.

2.2 Code Generation Benchmarks

LLM-based code-generation benchmarks have expanded from correctness to efficiency evaluation. HumanEval (Chen et al., 2021) evaluates function-level generation through program-based testing; KernelBench (Ouyang et al., 2025) transforms PyTorch workloads into high-performance GPU kernels under correctness and performance requirements; FlashInfer-Bench (Xing et al., 2026) evaluates candidate generation, correctness, performance, and deployment-related processes for LLM-inference kernels. However, these benchmarks treat functions or individual kernels as the evaluation units and leave scheduling, synchronization, and resource management to fixed external runtimes. In megakernels, the evaluated method may generate these system-level components, so individual-kernel performance cannot fully characterize megakernel performance. Although KernelBench-Mega (Arledge, 2026) evaluates LLMs' ability to generate megakernels, its task scope is overly narrow(only 2 tasks) and it lacks multilevel system-level evaluation.

2.3 High-performance Coding Benchmarks

High-performance coding benchmarks increasingly evaluate tool use, code execution, and feedback-driven engineering. SWE-bench (Jimenez et al., 2024) uses real-world GitHub issues, MLAgentBench (Huang et al., 2023) evaluates autonomous machine learning experimentation, and MLE-bench (Chan et al., 2024) evaluates agents on 75 Kaggle competitions covering data processing, model training, and experiment execution. These benchmarks assess planning, tool invocation, code modification, and iterative optimization, whereas megakernel generation must additionally satisfy numerical correctness, single-work-kernel structural requirements, and competitive end-to-end performance. They therefore cannot directly evaluate high-performance megakernel construction.

3 MegaKernelBench Design

MegaKernelBench evaluates a method's ability to generate high-performance megakernels for a specific model and hardware platform. This section defines the task types, workloads, hardware platforms, and evaluation protocol. Figure 1 summarizes the benchmark workflow and the progressively expanding generation scope of Tasks A–D.

3.1 Megakernel Definition and the MegaKernelBench Framework

A megakernel fuses multiple operators into a single persistent kernel and manages their execution through internal scheduling. Existing megakernel systems can generally be characterized by computation and scheduling. For example, Mirage Persistent Kernel (MPK) represents a megakernel as an SM-level computational task graph, where task nodes define computation and scheduling specifies their dependencies, execution order, and assignment across SMs. AutoMegaKernel similarly separates agent-driven optimization into an instruction debugging loop for operator optimization and a scheduling search loop. Accordingly, we view a megakernel as a system running on a lower-level runtime with two core components: the operator side and the scheduling side. The operator side performs inference computations, while the scheduling side determines task partitioning, execution order, SM mapping, dependencies, synchronization, and buffer access. Together, they determine end-to-end megakernel performance.

For a given inference workload, we define four categories of generation and optimization: **Operator-side generation and optimization:** The scheduling strategy remains fixed while operator implementations are generated and optimized. **Scheduling-side generation and optimization:** Operator implementations remain fixed while task partitioning, SM mapping, execution order, depen-

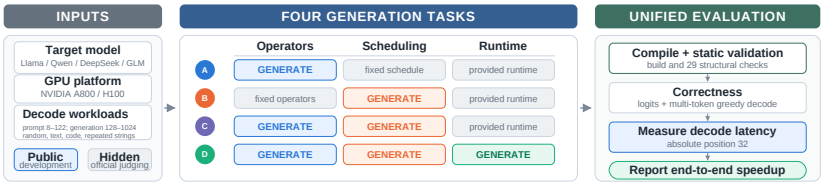


Figure 1: MegaKernelBench design: inputs, task scopes, and unified evaluation.

dencies, synchronization, and buffer access strategies are generated and optimized. **Joint operator-scheduling generation and optimization:** Operator implementations and scheduling strategies are jointly generated and optimized to maximize overall system performance. **Generation without a prebuilt runtime:** The evaluated method must generate the megakernel runtime itself from the target model and hardware specification, extending the capability hierarchy from component-level to system-level generation. The four task types correspond one-to-one to these four capabilities.

MegaKernelBench is built on AutoMegaKernel, a representative agent-based framework that separates operators, scheduling strategies, and the runtime. Its runtime follows a representative static scheduling paradigm: computational tasks are encoded as instructions, synchronization uses dependency counters, and the scheduling strategy is determined before execution. Tasks A–C retain its runtime and data structure definitions to provide unified and verifiable interfaces. This choice provides common evaluation infrastructure rather than imposing AutoMegaKernel on all megakernels, and tasks A–C do not provide implementation details of the AutoMegaKernel framework to avoid impact of the framework itself on evaluated methods; Task D does not restrict runtime design.

3.2 Task Types and Contents

MegaKernelBench defines four progressively more comprehensive tasks, with the reference implementation's operators and scheduling program both produced by an agent assisted by the AutoMegaKernel framework. Tasks A–C provide a fixed runtime and evaluate operator-side generation, scheduling-side generation, and their joint generation, respectively. Task D additionally includes the runtime in the generation scope. All tasks measure speedup over a specified baseline subject to correctness and structural validity requirements.

Task A: Operator-side generation and optimization. Task A evaluates the ability to generate operator implementations that maximize end-to-end megakernel performance under a fixed scheduling strategy and prebuilt runtime. Unlike single-operator benchmarks, it requires the evaluated method to optimize all operators as a system, considering the schedule and available hardware resources rather than individual operator performance alone. The benchmark provides each operator's interface contract, including function definitions and buffer read/write rules, PyTorch (Ansel et al., 2024) reference semantics, and public shape information. The evaluated method submits CUDA implementations covering all operators in the forward computation graph, which are assembled with the reference scheduling program for evaluation.

Task B: Scheduling-side generation and optimization. Task B evaluates the ability to generate a scheduling strategy under fixed operator implementations and a prebuilt runtime. The evaluated method may modify task partitioning, SM mapping, execution order, dependencies, synchronization, dependency-counter operations, and buffer access strategies. The benchmark provides the reference operator library, computational graph construction script, model structure, computational instruction information, and format-checking tools. The submitted scheduling program must pass structural validation for errors such as deadlocks and data races before being assembled with the reference operator library for evaluation.

Task C: Joint operator-scheduling generation and optimization. Task C evaluates joint optimization of both components on the prebuilt AutoMegaKernel runtime. It provides all information available in Tasks A and B. The evaluated method submits CUDA implementations for all operators and a scheduling program, which are assembled into a megakernel for end-to-end evaluation.

Task D: Generation without a prebuilt runtime. Task D evaluates the ability to generate a decode megakernel from scratch, including its operators, scheduling strategy, and runtime. The benchmark

fixes the prefill procedure and initial KV-cache construction but provides no prebuilt megakernel decode runtime or framework constraints. It specifies only the model type and templates for three interface functions:**(1) load**: loads the model weights once; **(2) reset**: clears state from the previous sequence and receives the initial KV cache; **(3) step**: performs one incremental decode step. The evaluated method submits a directory whose entry file implements these interfaces, while its internal design remains unrestricted. During evaluation, each decode step must use exactly one work kernel for model forward computation before undergoing functional and performance evaluation. The differences among the four tasks are summarized in Appendix A.

3.3 WORKLOADS

MegaKernelBench covers more than 10 different model architectures, including Llama (Grattafiori et al., 2024), Qwen (Yang et al., 2025), DeepSeek (Liu et al., 2024), GLM (Z.ai, 2026a), and GPT-oss (Agarwal et al., 2025), on NVIDIA A800 and H100 GPUs. Its workloads are divided into hidden test workloads for official evaluation and public workloads for local development.

Each workload contains a token sequence with a prompt length of 8–122 tokens and a generation length of 128–1024 tokens. The content covers random tokens, natural language, source code, and repeated strings, accounting for the effect of input content on intermediate numerical values. Test workloads are sampled using recorded seeds to ensure reproducibility. The public set contains eight workloads per model and is distinct from the hidden test set, which determines the official correctness results. Correctness is evaluated using multiple content types and decoding positions. Performance is evaluated on workloads selected with a fixed random seed of 0.

3.4 UNIFIED EVALUATION PROTOCOL AND METRICS

Each submission undergoes 4 sequential correctness and structural validity checks:

(i) Compilation. The submitted code must compile successfully.

(ii) Static structural validation. The scheduling program must pass checks for 29 classes of structural errors, including deadlocks, data races, and dangling references.

(iii) Single-token numerical correctness. Submitted and reference logits are compared element-wise, and both maximum absolute and relative errors must remain within predefined tolerances.

(iv) Multi-token greedy-generation consistency. Consecutive generated tokens must match the reference greedy output. A submitted token is treated as an approximately equivalent floating-point outcome if it belongs to the reference top-5 candidates and its logit difference from the reference top-1 candidate is below a predefined threshold.

Task D additionally enforces the structural requirements of a valid megakernel. After these checks, we measure the single-step decode latency at absolute position 32 under a unified timing protocol and report speedup against per-task baselines: Tasks A and B against the reference implementation, and Tasks C and D against native PyTorch. The complete protocol, including the megakernel criterion, the timing procedure, and the speedup baselines, is described in Appendix B.

4 EXPERIMENTS

4.1 MAIN RESULTS: CAN EXISTING METHODS GENERATE CORRECT MEGAKERNELS?

4.1.1 EXPERIMENTAL SETUP AND BASELINE COMPARISON

We evaluate agents on six target models, each with four tasks, resulting in 24 model-task combinations. In the main experiments, each evaluated method is allowed 3 iterative optimization rounds. In each round, the method may test its implementation on the public workload set and improve it based on error messages and output comparisons. For each model, we run the given performance workload 3 times and report the average. All evaluations are conducted on NVIDIA A800 and H100 GPUs.

Table 1 reports the speedup over PyTorch native cached decode on Task C (single decode step at absolute position 32) for the general-purpose agent codex, KDA (NVIDIA Labs, 2026), and AKO (Luo

et al., 2026), each driven by DeepSeek-V4-Flash (DeepSeek-AI, 2026b), GLM-5.3 (Z.ai, 2026b), or GPT-5.6-Sol (OpenAI, 2026), with Mirage Persistent Kernel (MPK) as the compiler-based non-agent baseline examining whether agents discover structural optimizations beyond compilation.

We further conduct ablation studies on the effects of optimization rounds, sample count, and reasoning effort. Complete results for these studies are provided in Appendix C.

4.1.2 Agent Performance Across Task Types

Figure 2 reports the final speedup achieved by submissions that pass all correctness checks across the four task types. In most cases, agents generate complete or partial implementations satisfying the correctness requirements in Tasks A, B, and C. On average, the megakernels generated for Task C achieve the highest performance. In Task D, fewer than 10% of the submissions successfully produce a correct implementation, indicating that megakernel generation without a prebuilt runtime remains a primary challenge for existing methods.

4.2 Analyzing the Megakernel Optimization Space

We conduct a case study independent of the main experiments to compare operator-side, scheduling-side, and joint optimization for small models and batch size=1. Specifically, an agent consisting of Claude Code and GLM-5.3 performs three optimization rounds on the Llama-3.2-1B tasks.

4.2.1 Characteristics of Independent Operator and Scheduling Optimization

Many operators generated by the agent substantially outperform their reference implementations. GEMV operators, which account for more than 90% of the computational tasks, achieve an average speedup of $1.75\times$, while attention achieves up to $4.45\times$. Nevertheless, the end-to-end speedup is only $1.05\times$. First, because scheduling-side optimization is disallowed, faster task execution does not reduce waiting under the unchanged dependencies; waiting accounts for more than 70% of latency under the reference schedule. Second, attention constitutes only a small fraction of computation for the short performance workloads. Third, the reference GEMV already reaches 81% of the A800 peak memory bandwidth, preventing its isolated speedup from being fully realized end to end. Thus, operator-side optimization is constrained by scheduling efficiency, workload composition, and hardware resource limits.

Scheduling-only optimization can also produce substantial gains. In Task B, finer task partitioning increases parallelism and improves performance by 13% over the reference implementation. However, excessively fine-grained partitioning increases fixed per-task costs, including task acquisition and dependency-counter checks. Increasing task granularity and reducing the number of tasks by 25% yields a further 10% improvement. Selecting a task count close to, but not exceeding, an integer

		Agent (Codex)			AKO			KDA			
GPU	Track	DS	GLM	GPT	DS	GLM	GPT	DS	GLM	GPT	MPK
A800	Llama-3.2-1B	1.05	1.04	1.05	0.86	1.00	1.04	0.46	0.60	0.55	8.01
	Qwen3-0.6B	2.17	2.28	2.16	1.88	2.18	1.04	0.83	1.24	1.17	15.06
	Qwen3-30B-A3B	4.08	3.81	3.92	3.61	3.94	3.62	4.42	3.74	3.53	–
	DeepSeek-v2-Lite	0.83	1.03	1.07	0.80	1.00	0.79	0.55	0.88	0.86	–
	GPT-oss-20B	–	1.34	0.30	1.14	1.11	1.12	1.20	1.18	1.15	–
	GLM-4.7-flash	–	1.76	1.34	1.13	1.23	1.26	0.93	1.17	1.33	–
H100	Llama-3.2-1B	0.92	0.90	0.76	0.86	1.06	1.15	0.77	0.78	0.76	8.80
	Qwen3-0.6B	1.82	1.94	1.87	2.02	2.60	2.28	1.23	2.08	1.86	15.43
	Qwen3-30B-A3B	1.99	5.46	5.45	4.85	3.09	3.12	5.34	4.08	3.21	–
	DeepSeek-v2-Lite	0.77	0.77	0.83	1.65	1.87	0.77	0.38	1.49	0.81	–
	GPT-oss-20B	–	0.90	–	–	1.32	2.44	0.76	0.87	0.83	–
	GLM-4.7-flash	–	1.93	1.51	1.12	1.07	1.38	1.42	1.33	1.53	–

Table 1: Main results on Task C: speedup over PyTorch native cached decode (higher is better) of the single decode step at absolute position 32 on NVIDIA A800 and H100 GPUs for three agent harnesses, each driven by one of three LLMs (DS = DeepSeek-V4-Flash, GLM = GLM-5.3, GPT = GPT-5.6-Sol), and Mirage Persistent Kernel (MPK). "–" denotes no passing submission or an unavailable result.

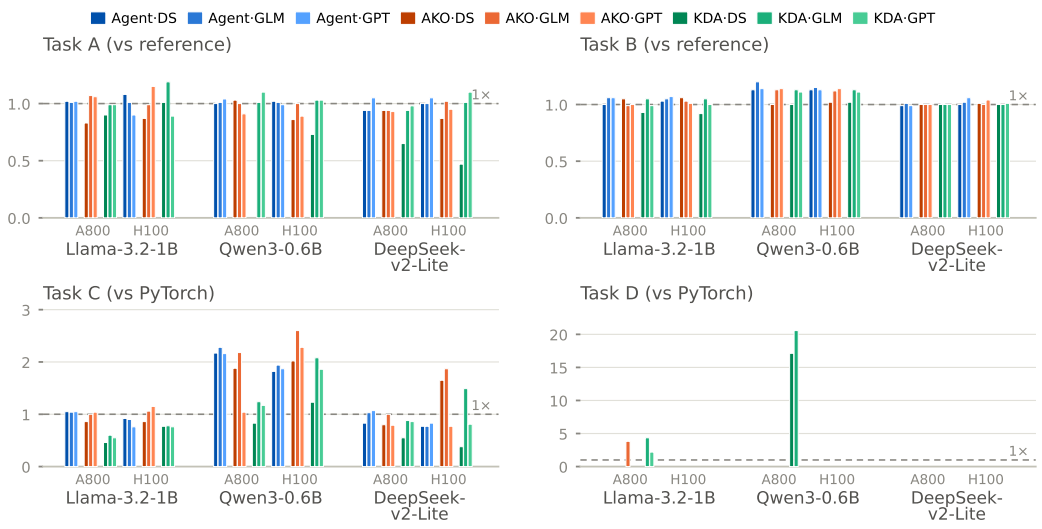


Figure 2: Final speedup of generated megakernels across the four task types (A–D), for each model track and evaluated LLM, computed over submissions that pass all correctness checks. Tasks A and B use the reference implementation as the baseline; Tasks C and D use PyTorch.

multiple of the number of SMs also avoids an additional execution wave and improves end-to-end performance by approximately 10%. Changing dependencies is more difficult. The agent fuses GEMV and ADD by connecting the value in the data buffer directly to the GEMV bias input and rewriting 2,080 task dependencies, achieving a 2%-3% improvement without modifying operators. However, most fusion opportunities require new operator variants. Scheduling-side optimization must therefore balance task granularity, parallelism, and fixed overhead, while dependency-only optimization offers limited opportunities when operator modifications are prohibited.

4.2.2 Independent Versus Joint Operator-Scheduling Optimization

Task C achieves a higher performance ceiling by allowing the scheduling strategy to be redesigned together with the operators. This enables changes to operator types, task organization, scheduling mechanisms, and cross-operator fusion. Retaining intermediate results in shared memory and prefetching weights across SMs provide additional but smaller gains.

K-split optimization for GEMV illustrates the need for coordinated modifications. It partitions computation along the K dimension across SMs, requiring operators to produce partial results and the scheduling strategy to adjust task partitioning, assignment, dependencies, and reduction. It therefore cannot generally be implemented by modifying either side alone.

Joint optimization also enables more extensive fusion. We fuse small operators such as ADD and RMSNORM into larger operators, reducing dependency levels from 13 to 7 per layer and from 211 to 115 across the model. Although the number of computational tasks decreases by only 3%, end-to-end performance improves by approximately 10%, saving nearly 1 ms per decode step. Fusion reduces dependency-counter handoff overhead by only 0.15 ms and eliminates only 0.9 MB/token of intermediate-result data movement, equivalent to 0.1% of the measured HBM traffic per step. Its primary benefit therefore arises from reducing the waiting caused by low-parallelism stages, rather than from reducing synchronization overhead or HBM traffic.

Simply combining independently optimized implementations does not achieve the same effect. On Llama-3.2-1B, Tasks A and B achieve speedups of $1.07\times$ and $1.23\times$, respectively, whereas their direct combination achieves only $1.26\times$, approximately 4% below their expected multiplicative gain. End-to-end latency is determined by the critical path of the task-dependency graph. Scheduling optimization redistributes tasks across SMs and changes this critical path, so operators that were bottlenecks under the original schedule may no longer dominate after scheduling optimization. Op-

erator and scheduling implementations must therefore be optimized jointly according to the resulting execution behavior.

4.2.3 MEGAKERNEL DECODE-LATENCY ANALYSIS

Megakernel decode latency consists of kernel launch overhead, computation overhead, and waiting overhead along the critical path. Computation overhead is the time during which SMs execute computational tasks. Waiting overhead is the time during which an SM is ready but unable to execute its next task because of hardware resource constraints, synchronization, or dependencies. It comprises dependency waiting determined by the scheduling strategy and overhead intrinsic to the runtime scheduling mechanism.

Table 2 presents the latency breakdown for Llama-3.2-1B. The reference megakernel launches one work kernel per decode step and incurs only 55 μs of launch overhead. PyTorch eager launches 677 kernels and incurs approximately 1,400 μs, while vLLM (Kwon et al., 2023) requires approximately 331 μs. Runtime scheduling and synchronization introduce 680 μs/token of overhead in the reference implementation. In contrast, critical-path computation takes 2.324 ms and dependency waiting reaches 9.884 ms. The primary optimization opportunities therefore lie in reducing critical-path computation and dependency waiting, corresponding to operator-side and scheduling-side optimization, respectively.

4.3 ANALYSIS OF AGENT ERRORS AND GENERATION BEHAVIORS

This section uses separate behavioral experiments with a larger modification budget than the three rounds used in the main experiments. We record every code modification, tool invocation, evaluation response, and submission result to obtain sufficiently long trajectories.

4.3.1 STATISTICAL ANALYSIS OF AGENT ERRORS

Most Task C failures arise from numerical accuracy issues, whereas compilation failures are rare. Stage-wise pass rates and the failure taxonomy are provided in Appendix D.

Error types vary substantially across tasks. In Task A, agents fail to generate functionally correct operators after the first iteration in 80% of cases. Single-step logits generally pass the numerical check, but top-1 token errors exceed the threshold during multi-token generation. Numerical errors are less frequent in Tasks B and C. Although Task C also requires operator generation, its lower numerical error rate suggests that jointly adjusting operators and scheduling may help avoid some numerical errors. In Task D, 80% of agents fail at static validation, and some submissions fail to compile. By contrast, with a prebuilt runtime, agents generally produce implementations that compile and satisfy the structural requirements, while numerical errors become the dominant source of failure. These results identify runtime generation as a major challenge that agents cannot resolve through a small number of iterations.

Category	Component	Duration (ms)	Share of total (%)
Critical-path computation		2.324	17.96
Critical-path waiting	Dependency waiting	9.884	76.36
	Runtime scheduling & synchronization overhead	0.680	5.25
Kernel launch		0.055	0.43
Total latency		12.943	100

Table 2: Decode-latency breakdown: duration and share of total latency per component.

Modification type	LLM	Round share (%)	Avg. score gain	Regr. (%)	Error (%)
Operator rewrite	DS-v4-Flash	6/10 (60.0%)	+0.0186×	1/6 (16.67%)	0/6 (0.0%)
	GLM-5.3	2/10 (20.0%)	-1.0590×	1/1 (100.0%)	2/2 (100.0%)
	GPT-5.6-Sol	4/10 (40.0%)	+0.3275×	2/3 (66.7%)	0/4 (0.0%)
Schedule reordering	DS-v4-Flash	1/10 (10.0%)	-0.9435×	1/1 (100.0%)	1/1 (100.0%)
	GLM-5.3	5/10 (50.0%)	-0.2205×	4/5 (80.0%)	1/5 (20.0%)
	GPT-5.6-Sol	5/10 (50.0%)	-0.3865×	4/5 (80.0%)	2/5 (40.0%)
Local fix	DS-v4-Flash	3/10 (30.0%)	+0.3529×	0/3 (0.0%)	0/3 (0.0%)
	GLM-5.3	3/10 (30.0%)	+1.1235×	0/3 (0.0%)	0/3 (0.0%)
	GPT-5.6-Sol	1/10 (10.0%)	+1.0240×	0/1 (0.0%)	0/1 (0.0%)

Table 3: Agent modification types and their effects during megakernel generation: share of modification rounds, average score gain, performance regression rate, and error rate, per modification type and LLM.

4.3.2 Statistical Analysis of Agent Generation Behaviors

We classify each modification as either a local fix or a substantive modification. Local fixes change masks, data types, data shapes, or similar details without altering computation or scheduling logic. Substantive modifications change operator algorithms, task partitioning or reordering, dependencies, or synchronization structures.

As summarized in Table 3, the two classes occur at similar frequencies. However, substantive modifications produce a negative average performance change per iteration, whereas local fixes almost always yield positive improvements. Later iterations generally provide small gains, and the performance difference between the first and final implementations is typically no more than 20%. Agents therefore possess knowledge of operator and scheduling optimization but rarely translate it into substantive modifications with positive performance gains.

Behavior also differs across task types. In Task A, implementations remain incorrect even after ten modification opportunities. In Task B, agents generally determine the dependency structure and task-partitioning granularity in the first iteration, then fine-tune granularity without substantial dependency changes or task repartitioning. Task C elicits a wider variety of techniques but remains dominated by local fixes. In Task D, only about 10% of the submissions generate a correct megakernel within ten modifications. The per-task distribution of failure stages is provided in Appendix E.

5 Limitations

Although MegaKernelBench provides a unified evaluation framework for megakernel generation and optimization, the current work has several limitations.

First, our evaluation focuses on single-GPU settings. Multi-GPU inference additionally involves inter-device communication, computation-communication overlap, and cross-device task scheduling. Consequently, the current benchmark cannot fully characterize megakernel generation and optimization for multi-GPU inference systems.

Second, the current benchmark primarily covers small-model inference with a batch size of one, which highlights the advantage of megakernels in reducing kernel launch overhead. For larger models and batch sizes, the contributions of computation and memory access to end-to-end latency may change. The current findings may therefore not generalize directly to these settings.

Third, the current tasks focus on static, relatively short-context decode workloads. Long-context inference, prefill, and dynamic batching may introduce more complex task dependencies and runtime scheduling requirements and are not covered by the current benchmark.

Finally, Tasks A-C use runtime of AutoMegaKernel as infrastructure. Its static execution paradigm that representing computational tasks as instructions and determining scheduling before execution matches the predominant design of current open-source megakernel systems but excludes dynamic scheduling and other future runtime paradigms. Task D imposes no restrictions on runtime design. MegaKernelBench can be extended to additional hardware platforms, model scales, and inference workloads for a more comprehensive characterization of the megakernel optimization space.

6 Conclusion

Experiments on MegaKernelBench lead to three main conclusions. First, operators and scheduling interact substantially. Simply combining independently optimized implementations loses part of their expected performance gains, whereas joint optimization supports more diverse strategies and a higher performance ceiling. Second, although existing methods can generate correct megakernels, the resulting performance gains remain modest. They tend to rely on local fixes and converge to correct but inefficient implementations, leaving high-performance megakernel generation an important challenge. Third, generation without a prebuilt runtime remains particularly difficult, requiring either reliable runtime infrastructure or efficient automatic runtime generation. Overall, MegaKernelBench extends evaluation from isolated kernels to complete megakernel systems and provides a unified foundation for studying their optimization space and generation methods across additional models and hardware platforms.

AI USE STATEMENT

Generative AI is used in this work in two capacities. First, as the object of evaluation: all generated megakernels reported in this paper were produced by LLM-based agents (codex agents driven by DeepSeek-V4-Flash, GLM-5.3, and GPT-5.6-Sol; the AKO and KDA harnesses; and a Claude Code + GLM-5.3 agent in the case study). Every correctness and performance number for these AI-generated artifacts was measured by our deterministic evaluation pipeline rather than by any model, and failure analysis combined automatic classification with manual inspection. Second, as a research assistant: parts of the benchmark's codebase (task construction, judging, and regression tooling) were implemented with the help of AI coding assistants and validated against reference implementations and adversarial negative cases; we also used AI assistants for manuscript formatting, for language polishing, and to help verify bibliographic entries, each of which was then checked against its primary source (arXiv, publisher pages, or official repositories) by the authors. The benchmark design, the evaluation-protocol design, and the analysis methodology were developed without generative AI, and no experimental numbers were produced by generative AI. We have reviewed all AI-assisted work and take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work involves no human subjects and no personal data. MegaKernelBench is built entirely on openly released model weights, which we use solely for research evaluation in accordance with their respective licenses. The benchmark is intended to support reproducible research on megakernel generation; submitted kernels are executed only within the benchmark's evaluation environments. We are not aware of harmful-insight risks or conflicts of interest beyond standard academic compute usage.

REPRODUCIBILITY STATEMENT

All evaluation artifacts will be released upon publication. The benchmark construction and task definitions are described in section 3, the workload design and seed-based sampling in section 3.3, and the correctness checks and timing protocol in section 3.4. All workloads are generated from recorded seeds, the performance workload uses a fixed seed of 0, and the public workload set allows local reproduction of the correctness pipeline, with per-track task details provided in Appendix F. Every reported latency follows the fully specified timing protocol: 10 warmup rounds, 4 alternating baseline/submission rounds, 30 measurements per round, and the average time of 4 rounds.

REFERENCES

Sandhini Agarwal et al. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*, 2025.

Osayamen Jonathan Aimuyo, Byungsoo Oh, and Rachee Singh. Flashmoe: Fast distributed moe in a single kernel. *arXiv preprint arXiv:2506.04667*, 2025.

Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 929–947, 2024.

Elliot Arledge. Kernelbench-mega: Full fused megakernel benchmarks for coding agents. `https://github.com/Infatoshi/kernelbench.com`, 2026.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, J. Aung, Devon Sherburn, Evan Mays, Gabriel Starace, et al. MLE-bench: Evaluating machine learning agents on machine learning engineering. *arXiv preprint arXiv:2410.07095*, 2024.

Mark Chen, Jerry Tworek, Heewoo Jun, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Xinhao Cheng, Zhihao Zhang, Yu Zhou, Jianan Ji, Jinchen Jiang, Zepeng Zhao, Ziruo Xiao, Zihao Ye, Yingyi Huang, Ruihang Lai, et al. MPK: A compiler and runtime for mega-kernelizing tensor programs. *arXiv preprint arXiv:2512.22219*, 2025.

DeepSeek-AI. Megamoe: fusing ep dispatch, gemms, swiglu, and ep combine into a single mega-kernel (deepgemm library). https://github.com/deepseek-ai/DeepGEMM, 2026a.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026b.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Hazy Research. Look ma, no bubbles! designing a low-latency megakernel for llama-1b. https://hazyresearch.stanford.edu/blog/2025-05-27-no-bubbles, 2025.

Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. MLAgentBench: Evaluating language agents on machine learning experimentation. *arXiv preprint arXiv:2310.03302*, 2023.

Jaber Jaber and Osama Jaber. Automegakernel: A statically-checked agent harness for self-retargeting megakernel synthesis. *arXiv preprint arXiv:2606.09682*, 2026.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*, 2024.

Hongyi Jin, Bohan Hou, Guanjie Wang, Ruihang Lai, et al. Event tensor: A unified abstraction for compiling dynamic megakernel. In *Proceedings of Machine Learning and Systems (MLSys)*, 2026.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, et al. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, pp. 611–626, 2023.

A Liu, B Feng, B Wang, et al. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*, 2024.

Mao Luo, Hongbin Li, Feng Lin, Hanling Yi, and Zhe Huang. Agentic kernel optimization: Generating state-of-the-art gpu kernels without hand-written cuda. *arXiv preprint arXiv:2608.14560*, 2026.

NVIDIA Labs. Kernel design agents (kda). https://github.com/NVlabs/kda, 2026.

OpenAI. Gpt-5.6: Frontier intelligence that scales with your ambition. https://openai.com/index/gpt-5-6, 2026.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? In *International Conference on Machine Learning (ICML)*, 2025.

Shanli Xing, Yi Zhai, et al. Flashinfer-bench: Building the virtuous cycle for ai-driven llm systems. *arXiv preprint arXiv:2601.00227*, 2026.

An Yang, Anfeng Li, Baosong Yang, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Z.ai. GLM-4.7-Flash. https://huggingface.co/zai-org/GLM-4.7-Flash, 2026a.

Z.ai. GLM-5.3. https://huggingface.co/zai-org/GLM-5.3, 2026b.

APPENDIX CONTENTS

A Task-Type Differences 13

B Evaluation Protocol Details 13

C Ablation Study Results 14

D Stage-wise Correctness Results 14

E Per-Task Failure Statistics 15

F Track Details 15

A TASK-TYPE DIFFERENCES

Table 4 summarizes the evaluation target, the information provided, the required submission, and the speedup baseline for each of the four task types. As noted in section 3.1, the AutoMegaKernel framework is adopted only because it cleanly separates operator-side from scheduling-side evaluation; the task statements themselves provide no information about how the framework is implemented.

	Task A	Task B	Task C	Task D
Evaluation target	Operator generation within a megakernel	Scheduling-strategy generation within a megakernel	Joint operator and scheduling generation within a megakernel	Megakernel generation without a prebuilt runtime
Information provided	Operator interface contracts, PyTorch reference semantics, model structure	Computational-graph construction script, model structure, computational-instruction information, format-checking tools	All information from Tasks A and B	Model type and interface function templates
Submission	CUDA implementations covering all operators of the forward pass	JSON scheduling program	A directory containing CUDA implementations of all operators and the JSON scheduling program	Megakernel implementation with its own runtime
Speedup baseline	Reference implementation	Reference implementation	PyTorch (reference implementation as auxiliary)	PyTorch

Table 4: Differences among the four task types.

B EVALUATION PROTOCOL DETAILS

Table 5 lists the parameters of the evaluation protocol described in section 3.4, covering numerical tolerances, the megakernel structural criterion, timing, and the judging scope.

Item	Value
First-step logits element-wise tolerance (bf16)	rtol $= 2 \times 10^{-2}$; atol calibrated empirically per model architecture
Maximum logits tolerance allowed on top-1 token mismatch	2.0
Megakernel criterion	exactly one work kernel per decode step; total kernel count (work plus non-work kernels) ≤ 8
Timing	alternating submission/baseline execution to suppress cold-start effects; 4 rounds with 30 measured executions each; the average latency is reported
Judging scope	correctness validation covers all tokens generated during decoding; performance measures the single-step decode latency at absolute position 32

Table 5: Parameters of the evaluation protocol.

Under the megakernel criterion, bookkeeping kernels must not participate in model computation. Kernel types are classified from profiler information; any kernel that participates in the model forward computation is classified as a work kernel.

After the correctness checks, we measure the single-step end-to-end decode latency at absolute position 32. Prompts longer than 32 tokens are truncated to the first 32 tokens, while a fixed PyTorch reference implementation greedily generates tokens for shorter prompts until the context length reaches 32. The reference implementation then performs prefill to construct the initial KV cache, from which the submitted implementation executes one decode step.

Each submission is warmed up for 10 rounds. During formal timing, the baseline and submitted implementation are executed alternately for four rounds; each round measures the decode step 30 times and reports its mean latency. Timing begins when the host launches the megakernel and ends when the full-vocabulary logits are produced. The same initial KV-cache state is restored before every execution, while state initialization and prefill are excluded from timing. The average latency among the four rounds is reported.

Tasks A and B use the reference implementation as the baseline and report end-to-end speedup for operator-side and scheduling-side optimization, respectively. Tasks C and D use native PyTorch as

the baseline to measure complete-megakernel performance relative to conventional inference. Task C additionally reports speedup over the reference megakernel, enabling joint-optimization analysis together with the Task A and B results.

C Ablation Study Results

This section reports the ablation studies summarized in section 4.1.1: the number of iterative optimization rounds (Table 6), the number of samples (Table 7), and the reasoning effort (Table 8).

Model	Iteration rounds	DS-v4-Flash	GLM-5.3	GPT-5.6-Sol
Llama-3.2-1B	1	1.1685×	0.9665×	0.6584×
	2	1.2014×	1.0859×	0.8136×
	4	1.2533×	0.9648×	0.9818×
	8	0.7900×	0.9326×	0.9935×
Qwen3-30B-A3B	1	3.0486×	2.0295×	3.6503×
	2	3.3436×	1.8947×	3.5335×
	4	2.9840×	3.6064×	3.4449×
	8	3.6273×	2.8727×	2.1784×

Table 6: Performance under different numbers of iterative optimization rounds.

Model	Number of samples	DS-v4-Flash	GLM-5.3	GPT-5.6-Sol
Llama-3.2-1B	1	1.2830×	1.2245×	0.0000×
	2	1.2830×	1.2245×	1.1265×
	4	1.2830×	1.2611×	1.1265×
	8	1.2931×	1.2611×	1.1265×
Qwen3-30B-A3B	1	3.8933×	0.0000×	3.5865×
	2	3.8933×	3.8911×	3.6938×
	4	3.8933×	3.8911×	3.6938×
	8	4.4718×	3.8911×	4.2448×

Table 7: Performance under different numbers of samples.

Model	Reasoning effort	DS-v4-Flash	GLM-5.3	GPT-5.6-Sol
Llama-3.2-1B	low	0.9966×	0.9954×	0.7966×
	high	1.0632×	1.2274×	0.8732×
	max	1.2437×	0.8761×	1.0614×
Qwen3-30B-A3B	low	3.6136×	2.1776×	3.6958×
	high	2.8005×	3.3245×	2.7899×
	max	3.3303×	2.1973×	2.0673×

Table 8: Performance under different reasoning-effort levels.

As a reliability check, the judge re-times the factory reference implementation as a third arm in the same interleaved timing rounds as the submission; the agreement between this arm and a reference submission observed in our runs (within 1%) guards against clock drift and cross-run interference.

D Stage-wise Correctness Results

Table 9 reports the pass rates of the sequential correctness and structural validity checks on Llama-3.2-1B and Qwen3-30B-A3B for the three evaluated LLMs.

Model	LLM	Compilation	Static validation	Single-step logits	Multi-step top-1	Final pass rate
Llama-3.2-1B	DS-v4-Flash	5/5	5/5	5/5	5/5	5/5
	GLM-5.3	5/5	5/5	5/5	5/5	5/5
	GPT-5.6-Sol	5/5	5/5	2/5	2/5	2/5
Qwen3-30B-A3B	DS-v4-Flash	5/5	5/5	4/5	4/5	4/5
	GLM-5.3	4/5	5/5	3/5	3/5	3/5
	GPT-5.6-Sol	5/5	5/5	5/5	5/5	5/5

Table 9: Pass rates of the sequential correctness and structural validity checks on Llama-3.2-1B and Qwen3-30B-A3B for the three evaluated LLMs.

The taxonomy is the check sequence itself: each submission is classified by the first stage it fails, in the order compilation, static structural validation (29 error classes), single-token numerical correctness, and multi-token greedy-generation consistency, with the megakernel structural criterion additionally applied to Task D.

E Per-Task Failure Statistics

Table 10 reports the distribution of failure stages across the four task types.

Stage	Task A	Task B	Task C	Task D
Compilation failure	0/10 (0%)	0/10 (0%)	0/10 (0%)	2/10 (20%)
Static-validation failure	0/10 (0%)	0/10 (0%)	0/10 (0%)	7/10 (70%)
Single-step logits failure	2/10 (20%)	0/10 (0%)	0/10 (0%)	0/10 (0%)
Multi-step top-1 failure	6/10 (60%)	0/10 (0%)	0/10 (0%)	0/10 (0%)
Final pass	2/10 (20%)	10/10 (100%)	10/10 (100%)	1/10 (10%)

Table 10: Distribution of failure stages across the four task types.

F Track Details

This section documents the evaluation platforms, the per-track model provenance and operator sets, and the judging calibration.

Workload scale. Each track ships a public development set of 8 workloads sampled by workload generator v3 with recorded seed 1, and a hidden test set of 57 workloads sampled with recorded seed 0. Workload content covers random tokens, natural language, source code, and repeated strings; the performance workload is selected with recorded seed 0.

Hardware and software environment. Table 11 lists the hardware and software environment of the two evaluation platforms. The A800 platform is an NVIDIA A800-SXM4-80GB, which is identical to the A100-SXM4-80GB in every single-GPU specification relevant to this work (compute capability, SM count, HBM capacity, and peak HBM bandwidth); only the NVLink bandwidth is reduced, and no experiment uses multi-GPU communication.

Model tracks. Table 12 summarizes the 13 model tracks currently provided by the benchmark and Table 13 lists the operator set that each track requires for Tasks A and C. The tracks span dense-GQA, MHA, MLA, MoE, and hybrid linear-attention architectures, so the required operator sets grow from 8 operators for the dense tracks to 12 for the MLA-plus-MoE tracks. All tracks except Qwen1.5-MoE-A2.7B (fp32) are evaluated in bf16 with a uniform element-wise relative tolerance of 2×10^{-2} and a uniform near-tie window of 2.0 over the top-5 candidates; the absolute tolerance is calibrated per architecture against the measured bf16 noise band of the eager reference on that architecture.

Parameter	A800 platform	H100 platform
GPU model	NVIDIA A800-SXM4-80GB	NVIDIA H100 80GB HBM3 (SXM)
Compute capability	8.0	9.0
SM count	108	132
HBM capacity	80 GB (HBM2e)	80 GB (HBM3)
Peak HBM bandwidth	2,039 GB/s	3,350 GB/s
L2 cache	40 MB	50 MB
NVIDIA driver	580.105.08	590.48.01
CUDA runtime	12.4	13.1
PyTorch	2.6.0	2.6.0

Table 11: Hardware and software environment of the two evaluation platforms.

Track	Open-weight release	Architecture summary
Llama-3.2-1B	Llama-3.2-1B-Instruct	16 layers, hidden size 2048, dense FFN of 8192, GQA with 32 query and 8 KV heads of head dimension 64, RoPE
Qwen3-0.6B	Qwen3-0.6B	28 layers, hidden size 1024, GQA with 16 query and 8 KV heads of head dimension 128, per-head QK-norm
Qwen3-30B-A3B	Qwen3-30B-A3B-Instruct-2507	48 layers, hidden size 2048, GQA with 32 query and 4 KV heads of head dimension 128, per-head QK-norm; MoE in every layer: 128 routed experts, top-8, expert FFN 768, top-k normalization
DeepSeek-v2-Lite	DeepSeek-Coder-V2-Lite-Instruct	27 layers with a dense first layer, hidden size 2048; MLA with 16 heads (query head dimension $192 = 128 + 64$ RoPE, KV compression rank 512, value head dimension 128), YaRN; MoE: 64 routed experts, top-6, shared expert
GPT-oss-20B	gpt-oss-20b	24 layers, hidden size 2880, GQA with 64 query and 8 KV heads of head dimension 64; MoE: 32 routed experts, top-4, expert FFN 2880; vocabulary 201,088; natively MXFP4 weights
GLM-4.7-flash	GLM-4.7-Flash	47 layers plus one MTP layer (ignored in evaluation), hidden size 2048; MLA with low-rank query path (query rank 768, $192 + 64$ RoPE, value head dimension 256, KV compression rank 512); MoE: dense first layer, then 64 routed experts, top-4 sigmoid routing with auxiliary-free bias and renormalization, shared expert
Llama-3.2-3B	Llama-3.2-3B-Instruct	28 layers, hidden size 3072, dense FFN of 8192, GQA with 24 query and 8 KV heads of head dimension 128, RoPE
Qwen2.5-3B	Qwen2.5-3B-Instruct	36 layers, hidden size 2048, GQA with 8 KV groups (query-to-KV ratio 8), head dimension 128, biased Q/K/V projections
GLM-4-9B	GLM-4-9B-0414	40 layers, hidden size 4096, dense FFN of 13696, GQA with 32 query and 2 KV heads of head dimension 128, interleaved partial RoPE
DeepSeek-Coder-6.7B	DeepSeek-Coder-6.7B-Instruct	32 layers, hidden size 4096, dense FFN of 11008, full multi-head attention (32 query and 32 KV heads, head dimension 128), linear RoPE scaling
Qwen3.5-9B	Qwen3.5-9B	hybrid attention: 24 gated-delta-net linear attention layers and 8 gated GQA layers; hidden size 4096, FFN 12288, vocabulary 248,320
Moonlight-16B-A3B	Moonlight-16B-A3B-Instruct	27 layers with a dense first layer, hidden size 2048; MLA with direct projection (no query low-rank), 16 heads of query head dimension $192 = 128 + 64$ RoPE, KV compression rank 512, value head dimension 128, plain RoPE; MoE: 64 routed experts, top-6, shared expert
Qwen1.5-MoE-A2.7B	Qwen1.5-MoE-A2.7B	24 layers, hidden size 2048, full multi-head attention (16 query and 16 KV heads of head dimension 128, biased projections); MoE: 60 routed experts, top-4 without top-k renormalization, shared expert, sigmoid gate; fp32 track

Table 12: The 13 model tracks of MegaKernelBench: open-weight releases and architecture summary.

Track	Required operator set (Tasks A and C)
Llama-3.2-1B	8 operators: `add`, `silu_mul`, `rmsnorm`, `embed`, `rope`, `kv_append`, `attention_tile`, `gemv_tile`
Qwen3-0.6B	8 operators: same set as Llama-3.2-1B
Qwen3-30B-A3B	11 operators: the Llama-3.2-1B set plus `router_topk`, `moe_gemv_tile`, `moe_combine`
DeepSeek-v2-Lite	12 operators: `add`, `silu_mul`, `rmsnorm`, `embed`, `gemv_tile`, `router_topk`, `moe_gemv_tile`, `moe_combine`, `mla_rope`, `mla_kv_append`, `mla_block_gemv`, `mla_attention`
GPT-oss-20B	11 operators: `add`, `rmsnorm`, `embed`, `rope`, `kv_append`, `attention_tile`, `gemv_tile`, `router_topk`, `moe_gemv_tile`, `moe_combine`, `oss_glu`
GLM-4.7-flash	12 operators: same set as DeepSeek-v2-Lite
Llama-3.2-3B	8 operators: same set as Llama-3.2-1B
Qwen2.5-3B	8 operators: same set as Llama-3.2-1B
GLM-4-9B	8 operators: same set as Llama-3.2-1B
DeepSeek-Coder-6.7B	8 operators: same set as Llama-3.2-1B
Qwen3.5-9B	10 operators: the Llama-3.2-1B set plus `gdn_conv`, `gdn_scan`
Moonlight-16B-A3B	12 operators: same set as DeepSeek-v2-Lite
Qwen1.5-MoE-A2.7B	11 operators: the Llama-3.2-1B set plus `router_topk`, `moe_gemv_tile`, `moe_combine`

Table 13: Per-track required operator sets for Tasks A and C. Dense tracks require 8 operators, the hybrid linear-attention track 10, MoE tracks 11, and MLA-plus-MoE tracks 12.