# QiMeng-MegaKernel: LLMs Optimizing LLM Inference via Automated MegaKernel Generation

**Weiqiang Xiong**[1,2] **Shaohui Peng**[1*] **Wenyi Li**[1,2] **Hao Lu**[2,3]
**Qirui Zhou**[2,3] **Congying Ma**[1,2] **Ziming Ye**[1,2] **Zhenyu Yi**[1,2]
**Yunji Chen**[2,3] **Qi Guo**[3] **Ling Li**[1,2*]
[1]Intelligent Software Research Center, Institute of Software, CAS, Beijing, China
[2]University of Chinese Academy of Sciences, Beijing, China
[3]SKL of Processors, Institute of Computing Technology, CAS, Beijing, China
xiongweiqiang20@mails.ucas.ac.cn, {pengshaohui,liling}@iscas.ac.cn

## Abstract

High-performance inference for large models is critical in latency-sensitive applications. Existing inference engines suffer from sequential execution of hundreds of fine-grained kernels, causing kernel launch overheads and redundant memory accesses. A promising solution is fusing multiple kernels into a single large kernel, namely MegaKernel. However, existing MegaKernel implementations are tightly coupled to specific models and GPU architectures and require extensive manual engineering. We propose an LLM-driven framework QIMENG-MEGAKERNEL, which leverages an Instruction-Centric Fusion Abstraction to decompose MegaKernels into tractable fusion segments, thereby enabling automated generation. Our framework jointly performs fusion strategy reasoning and CUDA code generation while hierarchically verifying correctness. Extensive experiments demonstrate that QIMENG-MEGAKERNEL consistently outperforms state-of-the-art inference engines (up to 5.04× vLLM and 2.42× SGLang) and compilers (up to 1.92× MPK), while supporting a broader range of models (dense, MoE, and VLA) and NVIDIA GPU platforms (Hopper, Ampere, and Ada).

## 1 Introduction

In latency-sensitive applications, such as embodied robots, on-device assistants, and real-time interactive systems, the inference efficiency of large models (LMs), including large language models (LLMs) and vision–language–action (VLA) models, is crucial, as it directly determines system responsiveness and user experience. Despite the rapid advancement of GPU computing capabilities, achieving high-performance LM inference remains a major challenge. A key bottleneck is that existing high-throughput and concurrency-oriented deployment systems (e.g., vLLM [19], SGLang [34], and TensorRT-LLM [24]) decompose inference into hundreds of fine-grained kernels executed sequentially. This paradigm results in excessive kernel launch overheads and frequent redundant memory accesses. For example, in Llama3.2-1B inference on A100, kernel launch overhead accounts for up to 41.8% of the total execution time (see Appendix B).

A promising solution to this bottleneck is the **MegaKernel** paradigm, which fuses model inference into a single large kernel. As shown in Figure 1(top-left), MegaKernels eliminate launch-induced bubbles and retain data in on-chip memory, substantially reducing off-chip memory traffic. While recent efforts demonstrate the feasibility of large-scale fusion, existing approaches—whether manual or compiler-based—face severe scalability and portability limitations. Manual implementations

*Corresponding authors.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

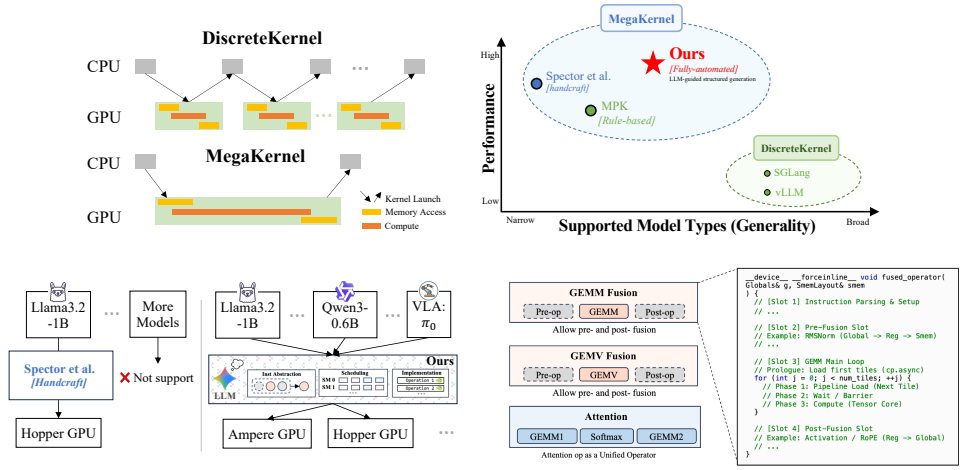


Figure 1: **Top-Left:** Advantages of MegaKernels; **Top-Right:** The generality–performance of existing approaches; **Bottom-Left:** Handcrafted vs. automated MegaKernel deployment; **Bottom-Right:** Instruction-Centric Fusion Abstraction and Optimization Skeletons.

(e.g., Spector et al. [29]) achieve strong performance but require prohibitive engineering efforts and deep low-level GPU expertise. Because they manually entangle fusion, scheduling, and synchronization logic, these large codebases are tightly coupled to specific models and hardware, incurring substantial migration costs. Conversely, compiler-based methods (such as MPK [8], which involves 100K lines of code) improve automation but are heavily constrained by carefully predefined transformation rules, rigid runtime mechanisms, and limited optimization spaces, leading to sub-optimal performance and restricted model support. Consequently, fully automating high-performance MegaKernel generation across diverse LM architectures and GPU platforms remains a critical challenge.

With the rapid advancement of LLMs for automated code generation, especially on high-performance kernels [11, 14, 22, 6, 20], a natural question arises: can we directly leverage LLMs to close this automation gap for MegaKernel deployment? Our empirical observations (see Appendix C) reveal that direct generation by LLM-based coding agents remains fundamentally insufficient. These failures correspond to **three critical challenges**: ***C1. Fusion Strategy:*** Persistent MegaKernel execution requires reasoning about global operator dependencies, fusion granularity, and task scheduling. LLMs struggle to perform explicit global fusion reasoning natively, frequently falling back to single-kernel implementations with sequential host-side launches. ***C2. Efficient Optimization:*** Effective MegaKernels must intricately coordinate computation pipelines, memory movement, synchronization, and resource utilization, which are tightly coupled and extremely difficult to infer accurately from fusion strategy alone. ***C3. Code Implementation Complexity:*** MegaKernel implementations typically consist of thousands of lines of CUDA code, which is large-scale and highly complex code that exceeds the capability of existing LLMs.

To address these challenges, we propose QiMeng-MegaKernel, an LLM-driven framework for automated high-performance MegaKernel generation across various LMs on diverse GPU platforms. At the core of QiMeng-MegaKernel is the **Instruction-Centric Fusion Abstraction**, which decouples the complex, monolithic MegaKernel generation process into multiple manageable semantic segments, referred to as **instructions**. By doing so, the abstraction makes both fusion strategy reasoning and low-level optimization tractable for LLMs, guiding the following **two-level generation framework:** ***Strategy Level (Addressing C1):*** Guided by the rules of our abstraction, the LLM automatically partitions the target model to instantiate fusion instructions. It then reasons out an efficient instruction schedule and layout mapping constrained by available hardware resources. ***Implementation Level (Addressing C2 & C3):*** To bridge the gap between strategy and implementation, the LLM refers to general optimization skeletons defined by the abstraction. This enables the LLM to reliably generate high-performance, functionally correct CUDA implementations for each instruction without being overwhelmed by code scale. Furthermore, QiMeng-MegaKernel introduces a hierarchical verification mechanism to rigorously validate both the generated fusion strategies and the underlying implementation, closing the loop for automated refinement.

In summary, our contributions are threefold: **(1)** We introduce an **Instruction-Centric Fusion Abstraction**, which decouples the complex full MegaKernel generation into multiple fused compo-

nents implementations. **(2)** We propose an **automated framework** that leverages LLMs for joint strategy reasoning and CUDA code generation to implement MegaKernel deployment. **(3)** Extensive results show that inference efficiency of MegaKernel automatically generated by QIMENG-MEGAKERNEL **consistently outperforms SOTA manually customized kernels**, inference engines (up to 5.04× vLLM and 2.42× SGLang) and compiler (up to 1.92× MPK). Besides, QIMENG-MEGAKERNEL can apply to more LMs (dense, MoE, and VLA) on more NVIDIA GPU platforms (Hopper, Ampere, and Ada).

## 2 Instruction-Centric Fusion Abstraction Mechanism

MegaKernel optimization fundamentally requires jointly reasoning about model semantics, hardware constraints, and low-level implementation details. The strong coupling of factors makes MegaKernel generation intractable for rule-based compilers and unreliable for end-to-end LLM generation.

Our key insight is that to make MegaKernel deployment tractable without sacrificing performance, the monolithic kernel must be decomposed into smaller, semantically meaningful segments. To this end, we introduce a generic *instruction-centric fusion abstraction* mechanism, which allows LLMs to use this abstraction to decompose a full MegaKernel into fused segments and instantiate them as instructions for scheduling, optimization, and implementation. Specifically, this mechanism captures key computational patterns in transformers, and associates each abstraction with an optimized implementation skeleton, as illustrated in Figure 1(bottom-right).

### 2.1 Instruction-Centric Fusion Abstraction

The Instruction-Centric Fusion Abstraction preserves the core operator types for each computational pattern, while abstracting away concrete tensor shapes and pre-/post-processing variants. Specifically, we identify three fundamentally different computational patterns in transformer-based LMs, each characterized by distinct compute behavior, hardware resource utilization, memory access patterns, and key optimization techniques.

**(1) GEMM-based Fusion Instruction** abstracts linear projection patterns whose execution is centered around high-throughput matrix multiplication, and are naturally suited for Tensor Core execution on modern GPUs, enabling high arithmetic intensity and efficient data reuse. **(2) GEMV-based Fusion Instruction** abstracts linear projection patterns whose execution is centered around matrix-vector multiplication. Its limited arithmetic intensity and irregular tile shapes are not suitable for Tensor Core execution, which may lead to wasted compute capacity and enforced alignment overhead. **(3) Attention Instruction** abstracts the complete attention computation as a single segment to reflect the unique semantics, facilitating holistic optimization and implementation(such as FlashAttention).

Through appropriate specialization, these abstractions can be composed to express a wide range of transformer-based LMs, ensuring representational completeness.

### 2.2 Optimization Skeleton

Beyond abstracting computational patterns, we further define an optimization skeleton for each instruction abstraction. It serves as an implementation-level abstraction that addresses the challenges of mapping fused instructions onto modern GPUs.

As illustrated in Figure 1(bottom-right), an optimization skeleton is organized as a set of structured slots that represent the whole execution workflow of a fusion instruction. These slots specify the data-movement and computation flow across the GPU memory hierarchy, including global-memory accesses, shared-memory staging, and register-level computation. They also expose key optimization opportunities, such as software pipelining for overlapping computation with memory operations, hardware-aware data layouts, and specific instructions for exploiting Tensor Cores and TMA.

Given such an optimization skeleton and the Python reference, the LLM first reasons to judge operator semantics, dataflow, synchronization, and hardware requirements. Then the LLM generates CUDA code according to these constraints. This structured process enables automated low-level CUDA generation while improving the correctness and reliability (See Appendix D for details).

## 3 QIMENG-MEGAKERNEL

Constructing a high-performance MegaKernel requires jointly determining model-level fusion, workload decomposition, and instruction scheduling. This is challenging because modern transformer-based LMs contain dozens of layers with heterogeneous operators and diverse tensor shapes, while GPU architectures expose different resource limits, memory hierarchies, and hardware primitives. The resulting strategy space for fusion and scheduling is extremely large. Moreover, even after a feasible strategy is identified, implementing it efficiently still requires thousands of lines of low-level CUDA code that carefully exploits hardware specifications. The scale and hardware-specific nature make correct and efficient MegaKernel construction highly challenging.

To address the challenges above, we proposed the QIMENG-MEGAKERNEL, which leverages the fusion abstraction and optimization skeleton described in Sec. 2 and automatically transforms the LMs into optimized MegaKernels through strategy determination, kernel implementation, and closed-loop verification. As shown in Figure 2, starting from a Hugging Face model, the system progressively reasons about execution strategies and resource allocation, generates low-level CUDA codes, and validates correctness via consistency checks with automatically generated Python reference code. Workflow pseudocode is provided in Appendix F.

### 3.1 Decomposition and Configuration

QIMENG-MEGAKERNEL leverages LLMs to automatically decompose the high-level model into instruction-level units and analyze the configuration of each instruction, thereby guiding scheduling and low-level code generation, as shown in step 1 of Figure 2.

**Model Decomposition.** LLM first decomposes the model into instruction instances that align with the fusion primitives defined in Section 2. Structured computation patterns, such as normalization followed by linear projection, are identified from the model's forward logic and mapped to concrete instruction instances, while preserving the original data dependencies and execution semantics.

**Configuration Analysis.** While instruction instantiation determines *what* computation should be performed, it does not yet specify *how* the computation should be realized on concrete execution backends. To bridge this gap, each instruction is augmented with explicit configuration information. For every instantiated instruction, the LLM derives two complementary configuration streams that decouple algorithmic semantics from hardware-specific optimization. The algorithm-related configuration captures hardware optimization agnostic properties, such as tensor shapes and memory bindings, generating instruction computation granularity and memory bindings based solely on computation types and dimensions. The configuration directly governs the generation of Python reference implementations. In contrast, the GPU-related configuration specifies optimization parameters such as thread-block tiling and warp-level specialization, which later guide the CUDA code generator.

### 3.2 Instruction Scheduling

A MegaKernel enables explicit static control over execution order and resource usage, making it possible to construct a bubble-free execution pipeline. To realize this, QIMENG-MEGAKERNEL leverages LLM to perform instruction scheduling and layout mapping in two explicit stages (Figure 2 step 2).

**Scheduling** is performed based on the scale and characteristics of each model layer, together with the shape configuration inferred in the previous step. This process determines the types of instructions to be executed and the number of times each instruction is invoked. The model is then scheduled into a concrete instruction sequence by respecting the computational dependencies among layers.

**Mapping** aims to distribute the instruction sequence as evenly as possible across all SMs to eliminate execution bubbles during SM execution. This is achieved by estimating the execution time of each instruction, which jointly accounts for the capability of the SM and the instruction's computation and memory access costs, thereby generating an instruction layout to maximize the usage of SMs.

Together, scheduling and layout generation ensure that all SMs progress in a synchronized manner and remain continuously occupied throughout the kernel's lifetime, effectively mitigating tail latency and maximizing hardware utilization.

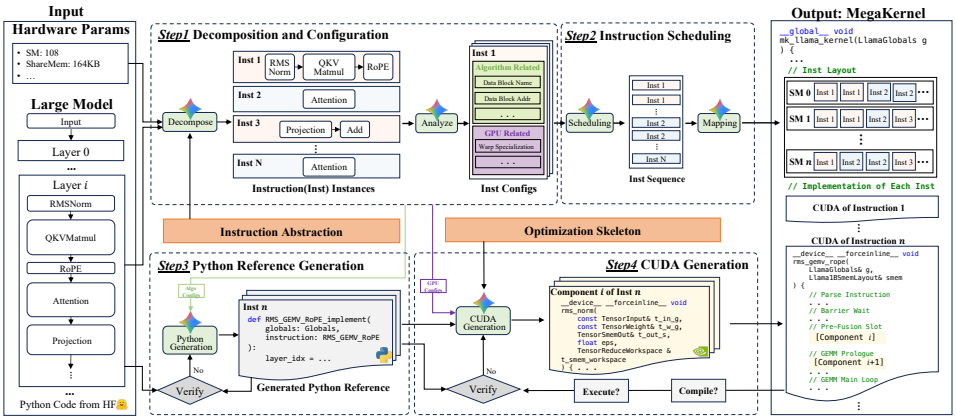


Figure 2: The **QIMENG-MEGAKERNEL** workflow consists of four stages: **(1) Decomposition and configuration.** Model layers are converted into instruction instances with analyzed configurations. **(2) Instruction scheduling.** Instruction sequences are mapped onto GPU SMs for optimized execution. **(3) Python reference generation.** An LLM generates executable Python code for correctness verification. **(4) CUDA generation.** Guided by skeletons and GPU configs, the LLM produces final CUDA kernels with a closed-loop verification.

### 3.3 Python Reference Generation

To establish a reliable semantic baseline before lowering to low-level CUDA code, the system generates an executable Python reference for each instruction. Guided solely by the algorithm-related configuration, the LLM generates a serial Python implementation that precisely captures the mathematical semantics, tensor shapes, and memory bindings of the target instruction, independent of hardware-specific details. This Python reference serves a dual role: it validates the correctness of instruction decomposition against the original HF model and provides an instruction-level reference for verifying the numerical correctness of the subsequently generated CUDA implementation.

### 3.4 Skeleton-guided CUDA Generation

Guided by the optimization skeletons defined for each abstraction in Section 2, the LLM sequentially generates the low-level CUDA implementation for each concrete instruction of the target model. Specifically, in addition to the skeleton itself, the generation process takes two key inputs: (1) the **Python implementation** of the corresponding instruction as reference code, which provides information about the operator types, compute granularity, and computational logic; and (2) **GPU-specific configuration information** for the instruction, which indicates hardware constraints such as warp specialization and the target GPU architecture.

Taking GEMM instructions as an example, the LLM follows the execution workflow specified by the skeleton and incrementally generates key components code, including memory access and computation logic for pre-operations, the core GEMM computation, software pipelining optimization and synchronous operation, data layout configurations for Tensor Core utilization, and the corresponding MMA instruction invocations. Then, the LLM encapsulates the complete implementation of each instruction as a '`__device__`' function, which serves as a building block for subsequent integration. Finally, the LLM assembles the separately generated device functions for instructions into a single '`__global__`' MegaKernel, followed by instruction layout produced at the strategy level (Code examples refer to Appendix E.2).

### 3.5 Hierarchical Verification

To guarantee the correctness of the generated MegaKernels without manual debugging, QIMENG-MEGAKERNEL employs a hierarchical verification mechanism.

**Stage 1.** Comparison of the Hugging Face (HF) Transformers implementation with our generated Python reference. As shown in Figure 3, a layer-by-layer check of intermediate values confirms that the generated MegaKernel strategy preserves the original model's behavior, validating the correctness of instruction decomposition and scheduling.

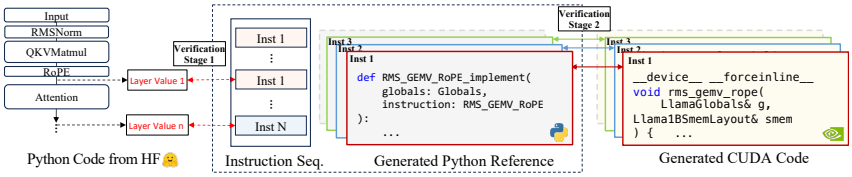


Figure 3: Hierarchical verification ensures correctness by aligning the generated Python reference code with the Hugging Face baseline and validating generated CUDA code numerically.

**Stage 2.** Comparison of each generated CUDA with its Python reference. Since the Python reference is derived from the same algorithmic configurations, it serves as an executable specification. As shown in the right part of Figure 3, our framework confirms that the numerical output of each CUDA function matches its Python reference.

**Iterative Refinement via Error Feedback.** Based on the verification feedback above, the pipeline employs a self-correction mechanism. When verification fails, the system initiates a refinement loop. It captures detailed diagnostic signals, such as nvcc error traces or tensor mismatch reports, and feeds them back to the LLM as structured feedback. Guided by this feedback, the LLM localizes the fault, reasons about the underlying cause, and regenerates the corrected strategy and code. This iterative generate–verify–refine cycle continues until the kernel passes all hierarchical checks, ensuring generation correctness without human intervention.

### 3.6 Theoretical Latency Analysis

In this section, we analyze how QIMENG-MEGAKERNEL reduces the execution latency compared with discrete-kernel with batch size $B$ and context length $L$:

$$\Delta T(B,L) = \underbrace{(K-1)\times \bar{t}_{\text{launch}}}_{(1)} + \underbrace{\left[\sum_{i=1}^{K}\left(T^{i}_{\text{dk_mem}}(B,L)+T^{i}_{\text{dk_exec}}(B,L)\right) - \left(T_{\text{mk_mem}}(B,L)+\sum_{j=1}^{K'}T^{j}_{\text{mk_exec}}(B,L)\right)\right]}_{(2)} + \underbrace{T_{\text{bubble}}(B,L)}_{(3)} - \underbrace{T_{\text{overhead}}(B,L)}_{(4)}.$$

The **term** $(1)$ comes from using a single MegaKernel generated by QIMENG-MEGAKERNEL to perform model inference, instead of relying on $K$ discrete kernels. With $\bar{t}_{\text{launch}}$ denoting the average launch overhead of a single kernel, $(K-1)\times\bar{t}_{\text{launch}}$ is the eliminated launch cost. This saving depends on the proportion of kernel launch overhead in the overall inference latency, and is particularly significant in lightweight inference scenarios.

The **term** $(2)$ comes from instantiating $K'$ fused instructions and executing them continuously on the GPU. Compared with $K$ discrete kernels, which incur repeated memory access and execution costs, the generated MegaKernel reduces redundant memory traffic by retaining intermediate results on-chip. This saving depends on the ratio between computation and memory access, and is more significant when $B$ and $L$ are small, where the arithmetic intensity is low.

The **term** $(3)$ comes from cross-SM scheduling of instruction instances. $T_{\text{bubble}}(B,L)$ denotes the execution bubbles reduced by scheduling and SM-level mapping. As $B$ and $L$ increase, the proportion of bubble overhead decreases, and this saving becomes less dominant.

The **term** $(4)$ is the negative effect introduced by MegaKernel overhead. $T_{\text{overhead}}(B,L)$ includes extra costs such as higher register pressure, larger shared-memory usage, and possible occupancy reduction. This overhead usually increases with $B$ and $L$.

Therefore, QIMENG-MEGAKERNEL is most advantageous for small-batch and short-to-medium-context decoding, where kernel launch overheads, redundant memory accesses, and execution bubbles dominate end-to-end latency, while the additional MegaKernel overhead remains limited.

## 4 Experiments

### 4.1 Evaluation Setup

To assess the generality and efficiency of QIMENG-MEGAKERNEL, comprehensive experiments are conducted with 6 different LMs, 3 hardware platforms, 6 SOTA inference engines and compilers.

**Target Models.** QIMENG-MEGAKERNEL is evaluated on six large models: 4 dense models (Llama3.2-1B, Llama3.2-3B, Qwen2.5-3B, Qwen3-0.6B), a MoE (Mixture-of-experts) model (Qwen1.5-MoE-A2.7B), and a VLA (Vision-Language-Action) model ($\pi_0$), spanning diverse attention mechanisms, layer structures, and model sizes.

**Hardware.** Experiments are conducted on diverse GPU architectures, including NVIDIA H100 (Hopper), A100 (Ampere) and L20 (Ada). (Detailed descriptions see Appendix A)

**Base LLMs and Agents.** We use Gemini-3-Pro [16] as the default base LLM and Codex as the agent framework for QIMENG-MEGAKERNEL across all experiments, with ablation studies on GPT-5.2 [26], Claude-Sonnet-4.5 [2], and DeepSeek-R1 [17]. Unless otherwise specified, the experiments use a temperature of 0.12, top-$p$ sampling with $p = 0.9$, and a maximum context length of 256k tokens.

**Baseline.** QIMENG-MEGAKERNEL is compared with three SOTA LLM inference engines (vLLM [19], SGLang [34], and Ollama [25]), NVIDIA's official library TensorRT-LLM [24], and recent MegaKernel-related works (human-crafted implementation Spector et al. [29] and the rule-based compiler MPK [8]). Notably, the above baselines do not support all evaluated models. For VLA $\pi_0$, the official optimized implementation (openpi) [4] and a manually fused implementation (Realtime-VLA, r-VLA) [23] are compared.

**Tasks.** For LLM inference, experiments are conducted on various batch sizes with varying output sequence lengths (128 to 1024 tokens), and six typical datasets (HumanEval [5], GSM8K [9], CoinFlip [31], MultiArith [28], ReClor [32], and SQuAD [27]). 200 samples are randomly selected from each dataset, all if fewer than 200 in the dataset. For VLA, we evaluate common scenarios with varying input views (1/2/3) and action chunk sizes (10/20/50). All performance results were measured while ensuring inference correctness.

**Metrics.** LLM inference performance is measured by throughput (tokens/s) used BF16 precision, while VLA performance is measured by the latency (ms) of the full output (action chunksize $\times$ 7-DoF). All results are averaged over 10 runs with 5 warm-ups. Raw data see Appendix J.

**Cost of generating MegaKernel.** See Appendix G.

### 4.2 Overall Performance

As shown in Figure 4 and Table 1, QIMENG-MEGAKERNEL automatically generates high-performance MegaKernel implementations across diverse model architectures and scales on H100 GPU, consistently outperforming all baselines across different output sequence lengths and evaluation datasets.

**QIMENG-MEGAKERNEL consistently generates MegaKernel implementation of the highest inference efficiency for multiple large models with different architectures and scales on GPUs**, without additional model-specific manual adaptation. In contrast, aside from general-purpose inference engines, other baselines exhibit varying degrees of incomplete support when applied to different model architectures or scales. These results validate that QIMENG-MEGAKERNEL, built upon an instruction-centric fusion abstraction, provides strong automation capability and generalization in generating efficient MegaKernel deployments across diverse large models.

**QIMENG-MEGAKERNEL consistently achieves substantial performance improvements over hand-crafted implementations, SOTA LLM inference engines, and compiler-based approaches.** Despite the extensive hand-engineered optimizations incorporated in SOTA inference engines, the MegaKernel generated by QIMENG-MEGAKERNEL consistently delivers strong performance gains, achieving speedups of up to 5.04$\times$ than vLLM, 2.42$\times$ than SGLang, 4.68$\times$ than Ollama, and 2.33$\times$ than the official library. Moreover, compared with the existing MegaKernel compiler MPK, QIMENG-MEGAKERNEL attains up to 1.92$\times$ inference speed. Notably, on representative models such as Llama3.2-1B and the VLA model $\pi_0$, QIMENG-MEGAKERNEL even surpasses fully hand-crafted implementations, including Spector et al.[29] and Realtime-VLA. These results indicate that, beyond enabling general-purpose MegaKernel generation via an instruction-centric fusion abstraction, QIMENG-MEGAKERNEL is capable of automatically identifying model-specific optimal task partitioning and scheduling strategies, while effectively exploiting hardware characteristics to achieve near-optimal inference performance. Section 4.4 further presents a detailed case study that breaks down the sources of performance gains over baselines.

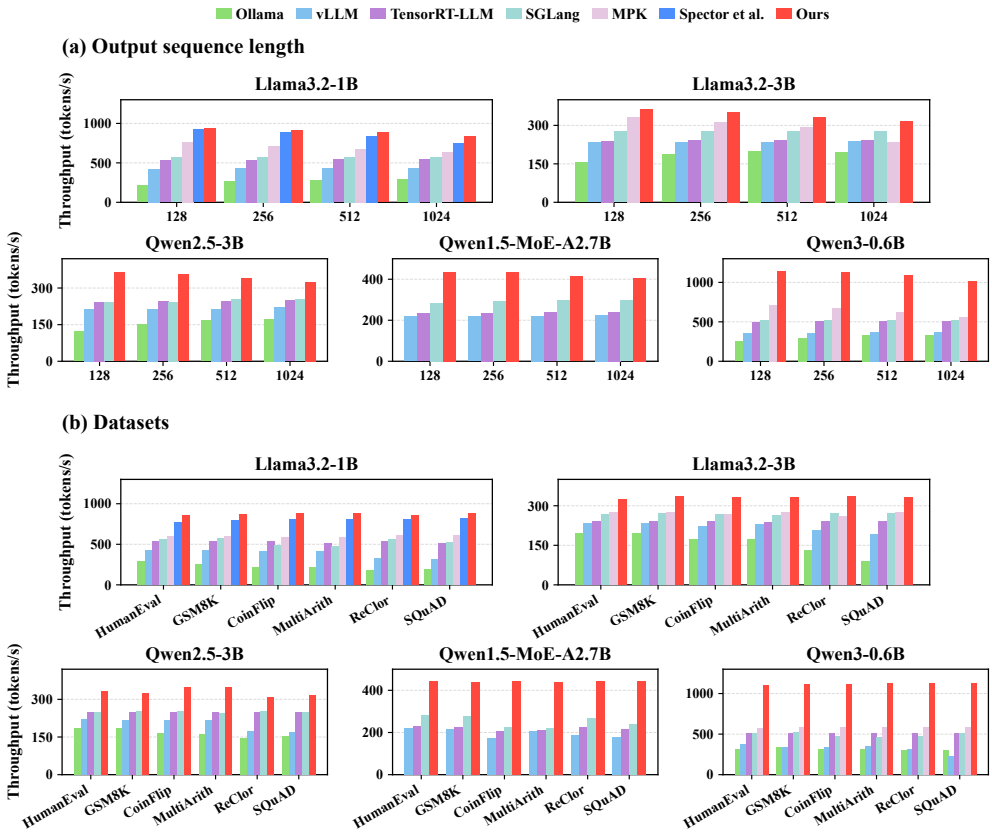


Figure 4: Inference throughput (tokens/s) of different LLMs and tasks on H100, varying in (a) output sequence length and (b) datasets. Missing bars indicate that the method does not support the corresponding model.

**QIMENG-MEGAKERNEL delivers consistent performance improvements across a wide range of sequence lengths and real-world evaluations.** We evaluate inference efficiency across output sequence lengths, and observe that QIMENG-MEGAKERNEL achieves strong performance gains at all tested lengths, especially for smaller models and shorter sequences. For example, on Qwen3-0.6B with a sequence length of 128 tokens, our method achieves up to 3.19× higher throughput compared to vLLM. In real-world evaluation workloads, QIMENG-MEGAKERNEL exhibits even more significant speedups, as the majority of questions typically involve short input (around 100 tokens). These results suggest that QIMENG-MEGAKERNEL is particularly well-suited for practical deployment scenarios and demonstrates strong potential for broad real-world applicability.

### 4.3 Ablation Study

**Ablation of target GPUs: Across multiple GPU generations (Hopper, Ampere, and Ada), QIMENG-MEGAKERNEL consistently achieves the best performance.** As shown in Figure 5, for the Llama3.2-1B model, QIMENG-MEGAKERNEL achieves speedup of up to 2.37× over inference engines. In contrast, some baseline methods exhibit performance degradation on certain GPU architectures; for example, vLLM and SGLang degrade on the L20 (Ada) GPU, falling behind Ollama. MPK relies on architecture-specific optimizations and cannot be directly applied to the Ada architecture. These results demonstrate the generalization of our instruction-centric fusion abstraction, together with the LLM-driven automated workflow, in reasoning about hardware- and parameter-specific strategies and generating efficient CUDA code across diverse GPU architectures. The code generated by QIMENG-MEGAKERNEL is included in Appendix E.

**Ablation of base LLMs: QIMENG-MEGAKERNEL can effectively drive different LLMs to generate high-performance MegaKernels.** As shown in Table 2, without QIMENG-MEGAKERNEL, only Gemini-3-Pro is able to generate correct but non-fused low-performance LLM inference code

Table 1: Performance comparison of VLA $\pi_0$ under different views and chunk sizes on H100, A100, and L20 GPUs. The lower latency (ms) is better.

| View | GPU | Chunk Size 10 | | | Chunk Size 20 | | | Chunk Size 50 | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Ours | openpi | r-VLA | Ours | openpi | r-VLA | Ours | openpi | r-VLA |
| 1 View | H100 | **15.70** | 21.61 | 16.22 | **16.20** | 38.60 | 16.70 | **16.39** | 39.63 | 17.29 |
| | A100 | **18.10** | 67.77 | 19.42 | **18.92** | 69.72 | 19.83 | **19.09** | 71.41 | 20.05 |
| | L20 | **22.36** | 94.12 | 23.70 | **39.71** | 95.29 | 40.61 | **39.81** | 99.14 | 40.71 |
| 2 Views | H100 | **19.02** | 26.90 | 19.61 | **19.42** | 39.42 | 19.97 | **19.63** | 40.31 | 20.00 |
| | A100 | **24.86** | 67.98 | 26.29 | **25.84** | 69.94 | 26.74 | **26.52** | 71.48 | 27.21 |
| | L20 | **31.99** | 94.30 | 33.32 | **42.89** | 95.23 | 44.50 | **44.17** | 98.97 | 45.26 |
| 3 Views | H100 | **21.28** | 31.79 | 22.08 | **21.80** | 39.68 | 22.38 | **22.26** | 40.41 | 22.95 |
| | A100 | **32.31** | 68.02 | 33.73 | **32.29** | 69.78 | 33.54 | **32.54** | 71.91 | 33.79 |
| | L20 | **43.16** | 94.10 | 44.37 | **55.89** | 95.20 | 57.78 | **56.21** | 98.79 | 58.74 |

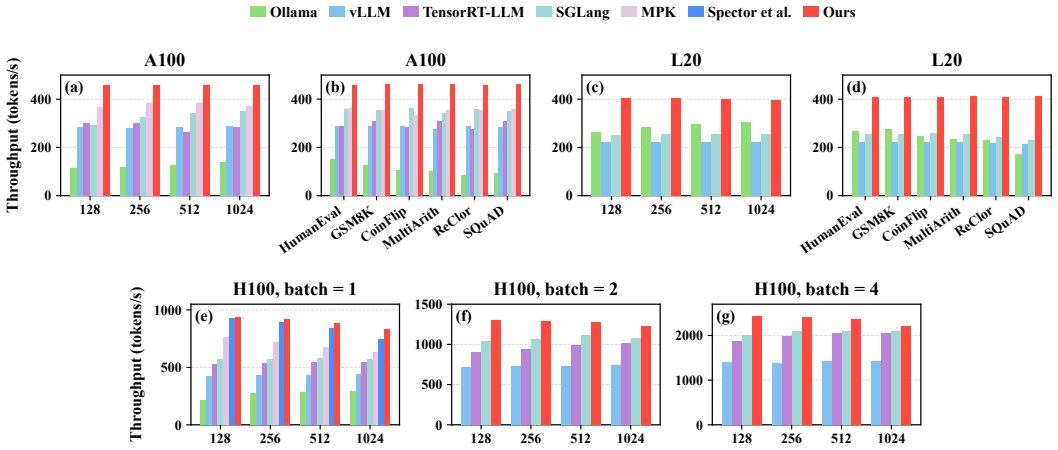


Figure 5: Performance of Llama3.2-1B on multiple GPUs (a-d), and under different batch sizes on H100 with varying output sequence lengths (e-g).

(90 tokens/s), while other models fail to produce LLM inference code. After applying QiMeng-MegaKernel, all evaluated LLMs are able to generate high-performance MegaKernels (at most 464.24 tokens/s). This observation suggests that QiMeng-MegaKernel decomposes the MegaKernel deployment task into manageable sub-stages that can be handled by base LLMs with varying capabilities, demonstrating the robustness of our approach.

**Ablation of batch size: QiMeng-MegaKernel maintains stable performance across batch sizes.** Besides the single-batch setting, we further evaluate its behavior under multi-batch configurations. As shown in Figure 5, compared to the baselines, QiMeng-MegaKernel maintains stable high performance as the batch size increases from 1 to 4, especially on small batch sizes and shorter sequence lengths. The reason is comprehensively analyzed in Section 3.6, and it further highlights the advantages of our generated MegaKernels in latency-sensitive scenarios.

### 4.4 Case Study

As shown in Table 3, we provide a detailed analysis of the per-token inference time distribution. The results are consistent with our expectations: (1) Compared to the Discrete Kernel paradigm in vLLM and SGLang, the MegaKernel generates only a single persistent kernel on the GPU, significantly reducing kernel launch overhead. Thanks to an effective soft pipeline, memory access costs are also reduced. (2) Compared to the rule-based MPK compiler, our approach performs strategy reasoning based on both the model and hardware, enabling more effective utilization of hardware resources and substantially reducing kernel execution time.

Table 2: Performance (token/s) under different base LLMs of Llama3.2-1B on A100 GPU.

| **Base LLMs** | **128** | | **256** | | **512** | | **1024** | |
|---|---|---|---|---|---|---|---|---|
| | Vanilla | +Ours | Vanilla | +Ours | Vanilla | +Ours | Vanilla | +Ours |
| Gemini-3-Pro | 88.97 | 458.82 | 87.76 | 458.66 | 90.47 | 456.93 | 88.57 | **456.69** |
| GPT-5.2 | ✗ | 448.45 | ✗ | 447.06 | ✗ | 444.04 | ✗ | 437.61 |
| Claude-Sonnet-4.5 | ✗ | 463.48 | ✗ | **461.86** | ✗ | 458.95 | ✗ | 452.21 |
| DeepSeek-R1 | ✗ | **464.24** | ✗ | 461.56 | ✗ | **459.49** | ✗ | 453.14 |

Table 3: Per-token inference time of Qwen3-0.6B on H100. Memory Access refers data transfer time between the host and device.

| Time(ms) | Ours | MPK | vLLM | SGLang |
|---|---|---|---|---|
| Kernel Number | 1 | 1 | 94 | 39 |
| Kernel Launch | **0.033(3.83%)** | 0.058(4.17%) | 0.352(12.61%) | 0.111(6.13%) |
| Kernel Execute | **0.828(96.17%)** | 1.334(95.83%) | 2.434(87.18%) | 1.696(93.70%) |
| Memory Access | **<0.001(<0.12%)** | <0.001(<0.08%) | 0.006(0.21%) | 0.003(0.17%) |
| Total | **0.861** | 1.392 | 2.792 | 1.810 |

# 5 Related Work

**Large Model Architectures.** LLMs include dense and MoE architectures, as well as derivative VLA models, each exhibiting distinct execution characteristics. Dense LLMs, such as Llama [30] and Qwen series [3], exhibit low arithmetic intensity during decoding and are primarily memory-bandwidth limited. MoE architectures [15] activate only a subset of experts per token, reducing FLOPs but introducing dynamic control flow, irregular memory access, and communication overheads, further amplifying decoding latency [21]. VLA models [35, 4] target embodied AI applications, where inference typically runs at batch size one under strict real-time constraints. Despite these differences, all three model families share a high-level execution pattern: latency-critical decoding composed of fine-grained, memory-bound operators, though inter-model variations in execution details make systematic MegaKernel generation challenging.

**LLM Inference Engines.** A growing ecosystem of LLM inference engines has been developed to improve serving efficiency, primarily targeting throughput and concurrency. vLLM [18] introduced PagedAttention to manage KV-cache efficiently and enable continuous batching, improving GPU utilization under high-concurrency workloads. SGLang [34] further optimized prefix reuse via RadixAttention and reduced runtime overheads through co-designed execution control. Production systems such as TensorRT-LLM [24] and LMDeploy [10] rely heavily on vendor-specific kernels and specialized operator libraries. These engines are typically built upon highly optimized operator libraries (e.g., FlashAttention [13, 12]), combined with additional control and orchestration logic. While this design is effective for compute-intensive phases such as prefill, it decomposes inference execution into hundreds of kernels. In latency-sensitive decoding scenarios—especially with lightweight models and small batch sizes—frequent kernel launches and global synchronization barriers dominate end-to-end latency, leading to poor hardware utilization.

**AI Compilers and MegaKernels.** AI compilers aim to automate performance optimization for tensor programs. Systems such as TVM [7], Ansor [33], and TorchInductor [1] apply heuristic- or search-based strategies to optimize individual operators or fuse small groups of adjacent operators. While effective for local optimizations, these compilers fundamentally operate at the operator or small-subgraph level and cannot generate large-scale inference for entire LLMs. Recent efforts [29] demonstrated a handcrafted MegaKernel for Llama-1B that achieves near-optimal decoding latency, but requires substantial manual engineering. MPK [8] proposes a compiler to automatically generate MegaKernels via an SM-centric intermediate representation, but it targets specific models and hardware and still relies on over 100K lines of expert-written CUDA and C++ code. Despite these advances, existing approaches still struggle to automate high-performance MegaKernel inference deployment across various LLM architectures on diverse GPU platforms.

# 6 Conclusion

We present QIMENG-MEGAKERNEL, an LLM-driven framework for automated MegaKernel generation. Through an Instruction-Centric Fusion Abstraction, it enables LLM joint reasoning over fusion strategies and correct code implementation under the specific model and hardware context. Extensive evaluations show our framework consistently promotes LMs' inference efficiency across dense, MoE, and VLA LMs on multiple generations of NVIDIA GPUs. By enabling LLMs to optimize their own kernels, our approach offers the potential for autonomous self-improving AI systems.

**Limitations and Future Work.** Given the dominance of NVIDIA GPUs in LLM inference, we verified QIMENG-MEGAKERNEL on three representative generations (Hopper, Ampere, and Ada). Future work will extend QIMENG-MEGAKERNEL to broader hardware platforms, including diverse GPU vendors and AI accelerators.

## Acknowledgments and Disclosure of Funding

This work is supported by the Strategic Priority Research Program of the Chinese Academy of Sciences (Grant No. XDA0700203, XDA0700200, XDA0700000).

## References

[1] 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*, pages 929–947, 2024.

[2] Anthropic. Introducing claude sonnet 4.5, September 2025. URL `https://www.anthropic.com/news/claude-sonnet-4-5`. Accessed: 2026-01-20.

[3] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

[4] Kevin Black, Noah Brown, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolo Fusai, Lachy Groom, Karol Hausman, Brian Ichter, et al. $\pi_0$: A vision-language-action flow model for general robot control. *arXiv preprint arXiv:2410.24164*, 2024.

[5] Mark Chen. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[6] Terry Chen, Zhifan Ye, Bing Xu, Zihao Ye, Timmy Liu, Ali Hassani, Tianqi Chen, Andrew Kerr, Haicheng Wu, Yang Xu, et al. Avo: Agentic variation operators for autonomous evolutionary search. *arXiv preprint arXiv:2603.24517*, 2026.

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

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

[9] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[10] LMDeploy Contributors. Lmdeploy: A toolkit for compressing, deploying, and serving llm. `https://github.com/InternLM/lmdeploy`, 2023.

[11] Weinan Dai, Hanlin Wu, Qiying Yu, Huan-ang Gao, Jiahao Li, Chengquan Jiang, Weiqiang Lou, Yufan Song, Hongli Yu, Jiaze Chen, et al. Cuda agent: Large-scale agentic rl for high-performance cuda kernel generation. *arXiv preprint arXiv:2602.24286*, 2026.

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

[13] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

[14] He Du, Qiming Ge, Jiakai Hu, Aijun Yang, Zheng Cai, Zixian Huang, Sheng Yuan, Qinxiu Cheng, Xinchen Xie, Yicheng Chen, et al. Kernel-smith: A unified recipe for evolutionary kernel optimization. *arXiv preprint arXiv:2603.28342*, 2026.

[15] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.

[16] Google DeepMind. Gemini 3 pro, 2026. URL https://deepmind.google/models/gemini/pro/. Accessed: 2026-01-20.

[17] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[18] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626, 2023.

[19] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention, 2023. URL https://arxiv.org/abs/2309.06180.

[20] Gang Liao, Hongsen Qin, Ying Wang, Alicia Golden, Michael Kuchnik, Yavuz Yetim, Jia Jiunn Ang, Chunli Fu, Yihan He, Samuel Hsia, et al. Kernelevolve: Scaling agentic kernel coding for heterogeneous ai accelerators at meta. *arXiv preprint arXiv:2512.23236*, 2025.

[21] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[22] Wei Liu, Jiawei Xu, Yingru Li, Longtao Zheng, Tianjian Li, Qian Liu, and Junxian He. Dr. kernel: Reinforcement learning done right for triton kernel generations. *arXiv preprint arXiv:2602.05885*, 2026.

[23] Yunchao Ma, Yizhuang Zhou, Yunhuan Yang, Tiancai Wang, and Haoqiang Fan. Running VLAs at real-time speed. *arXiv preprint arXiv:2510.26742*, 2025.

[24] NVIDIA. Tensorrt-llm: A tensorrt toolbox for large language models. https://github.com/NVIDIA/TensorRT-LLM, 2023. Accessed: 2026-01-19.

[25] Ollama. Ollama: Get up and running with llama 3, mistral, gemma, and other large language models, 2023. URL https://github.com/ollama/ollama. Software available from GitHub.

[26] OpenAI. Introducing gpt-5.2, December 2025. URL https://openai.com/index/introducing-gpt-5-2/. OpenAI Blog, Accessed: 2026-01-20.

[27] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*, 2016.

[28] Subhro Roy and Dan Roth. Solving general arithmetic word problems. In *Proceedings of the 2015 conference on empirical methods in natural language processing*, pages 1743–1752, 2015.

[29] Benjamin Spector, Jordan Juravsky, Stuart Sul, Owen Dugan, Dylan Lim, Dan Fu, Simran Arora, and Christopher Ré. Look ma, no bubbles! Designing a low-latency megakernel for Llama-1b. Hazy Research Blog, may 2025. URL https://hazyresearch.stanford.edu/blog/2025-05-27-no-bubbles. Accessed: 2026-01-17.

[30] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[31] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[32] Weihao Yu, Zihang Jiang, Yanfei Dong, and Jiashi Feng. Reclor: A reading comprehension dataset requiring logical reasoning. *arXiv preprint arXiv:2002.04326*, 2020.

[33] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pages 863–879, 2020.

[34] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.

[35] Brianna Zitkovich, Tianhe Yu, Sichun Xu, Peng Xu, Ted Xiao, Fei Xia, Jialin Wu, Paul Wohlhart, Stefan Welker, Ayzaan Wahid, et al. Rt-2: Vision-language-action models transfer web knowledge to robotic control. In *Conference on Robot Learning*, pages 2165–2183. PMLR, 2023.

# Appendix

**A GPU Architectures** **14**

**B Kernel Launch Overhead Measurement** **15**

**C Evaluating the Ability of LLMs to Generate MegaKernels** **15**

C.1 Experimental Objective . . . . . 15

C.2 Input and Output Specification . . . . . 15

C.3 Failure Analysis . . . . . 17

C.4 Summary . . . . . 18

**D Example of an Optimization Skeleton** **19**

**E Examples of Generated CUDA Code** **20**

E.1 Generated Code for rms_qkv_gemv_rope Instruction . . . . . 20

E.2 Generated Code for rms_qkv_gemm_rope Instruction . . . . . 23

**F Overall Workflow of QIMENG-MEGAKERNEL** **25**

**G Generation Cost of MegaKernel** **26**

**H Detailed Latency Analysis** **27**

**I Vanilla Prompt for Ablation Studies** **29**

**J Raw Measurement Data for Overall Performance** **29**

## A GPU Architectures

This section summarizes architectural features of modern NVIDIA GPUs, focusing on the differences between the Ampere, Ada and Hopper architectures, which are most relevant to kernel optimization.

**Ampere Architecture.** The Ampere architecture (e.g., A100) introduced third-generation Tensor Cores, enabling high-performance mixed-precision computation across a wide range of data types, including FP16, BF16, TF32, and INT8. Tensor Core operations in Ampere are primarily executed via warp-synchronous MMA instructions. Furthermore, Ampere supports asynchronous data transfers from global to shared memory, facilitating software-managed pipelining through shared-memory staging to effectively overlap data movement with computation.

**Ada Architecture.** The Ada architecture (e.g., L20) largely preserves the programming model of Ampere while improving cache capacity and SM microarchitecture. These changes make performance highly sensitive to memory layout and access patterns, including contiguity, alignment, and coalescing, which directly affect effective bandwidth utilization. Unlike Hopper, Ada does not provide warp-group asynchronous MMA or a dedicated tensor data-movement engine such as TMA. Therefore, overlap between computation and memory movement still relies on software-managed pipelining based on `cp.async` and staged shared-memory buffering, together with explicit synchronization. As a result, kernel performance remains strongly dependent on the balance among register usage, occupancy, and instruction-level parallelism.

**Hopper Architecture.** The Hopper architecture (e.g., H100) significantly extends Ampere by introducing fourth-generation Tensor Cores, new data formats such as FP8, and more advanced hardware support for asynchronous execution and data movement. Another innovation in Hopper is Warpgroup MMA(WGMMA), which decouples issue from completion and allows multiple matrix operations to be in flight concurrently. Hopper also introduces TMA, a hardware engine that offloads bulk, asynchronous, and multidimensional data transfers. By operating independently of warp execution, it significantly reduces the overhead of explicit data-movement orchestration within the kernel.

# B Kernel Launch Overhead Measurement

To measure kernel launch overhead during inference, we evaluate two implementations on Llama-3.2-1B with an NVIDIA A100 GPU: the decomposed MegaKernel generated by QIMENG-MEGAKERNEL and vLLM. We measure the number of launched operators, execution time, total launch time, average launch time per launch, and the launch-overhead ratio. The results are summarized in Table 4.

Table 4: Kernel launch overhead for Llama-3.2-1B inference on an NVIDIA A100 GPU.

| Implementation | # Operators launched | Execution time ($\mu$s) | Total launch time ($\mu$s) | Avg. launch time ($\mu$s) | Launch-overhead ratio (%) |
|---|---|---|---|---|---|
| Decomposed MegaKernel | 97 | 3760.00 | 1570.00 | 16.35 | 41.80 |
| vLLM | 63 | 3228.80 | 770.44 | 12.23 | 23.86 |

These results show that kernel launch overhead is substantial for both implementations. In particular, kernel launch overhead remains a major bottleneck, leaving considerable room for optimization.

# C Evaluating the Ability of LLMs to Generate MegaKernels

## C.1 Experimental Objective

The central question is whether LLMs can generate a MegaKernel without the structured scaffold provided by QIMENG-MEGAKERNEL . We evaluate this question under three increasingly favorable conditions for the LLM:

1. **Direct generation from a high-level performance goal.** The LLM is asked to generate a high-performance CUDA implementation for single-batch inference, with instructions to reduce kernel-launch overhead, use shared memory, and fused instructions. This experiment evaluates GPT-5.2, Claude-Sonnet-4.5, and DeepSeek-R1.
2. **Generation with the intended fusion strategy provided.** The LLM is explicitly given the intended fused execution structure and is asked to generate a MegaKernel-like implementation. This experiment uses GPT-5.2, which was selected based on the results of the first experiment.
3. **Generation with human-crafted MegaKernel reference.** The LLM is provided with a human expert MegaKernel implementation for one model and is asked to adapt it to another model. This experiment also uses GPT-5.2.

The first experiment evaluates the LLMs' natural generation tendency. The second tests whether the failure is mainly due to an underspecified prompt. The third tests whether providing expert code examples is sufficient for reliable MegaKernel generation or migration.

## C.2 Input and Output Specification

### C.2.1 Experiment 1: Direct High-Performance Decode Generation

**Input to the LLM.** Each LLM receives the target setting of single-batch decode for Llama3.2-1B on a modern NVIDIA GPU with bf16 computation, together with a high-level goal of high-performance inference. The prompt encourages reducing launch overhead, using shared memory, and applying operator fusion where appropriate.

**Evaluated models.** This experiment evaluates three LLMs: GPT-5.2, Claude-Sonnet-4.5, and DeepSeek-R1. The experiment is conducted using the agent framework Codex, running on NVIDIA A100.

**Expected output.** A successful output should be a complete CUDA/Python extension implementing one decode step. In particular, a successful MegaKernel implementation should contain persistent device-side execution, cross-operator fusion, a coherent host/device ABI, correct KV-cache semantics, and numerical agreement with a standard framework reference implementation.

**Observed output.** Across the evaluated LLMs, no model generated a persistent MegaKernel. The generated programs were either discrete-kernel pipelines or shallow local-fusion implementations. Some outputs could be made to compile and execute after engineering fixes, but none passed full correctness validation against the reference implementation.

At a high level, the generated structures fell into the following categories:

Table 5: Qualitative outcomes of direct high-performance decode generation.

| LLM | Generated structure | Validation result |
|---|---|---|
| DeepSeek-R1 | Partial local fusion | Attempt aggressive local fusion, but the generated multi-file interfaces are inconsistent and the implementation can not be reliably validated. |
| GPT-5.2 | Discrete pipeline with minor fusion | Run successfully and sometimes match the top prediction, but the logits error exceeds the required tolerance. |
| Claude-Sonnet-4.5 | Mostly per-op kernel pipeline | Run after fixes, but numerical error remained severe. |

These results show that, under natural high-performance prompting, LLMs tend to reproduce familiar CUDA programming patterns: standalone operator kernels, host-side sequential launches, and limited local fusion such as combining RoPE with cache update or fusing adjacent MLP operations. They do not spontaneously construct the system-level structure required by a persistent MegaKernel, such as opcode dispatch, SM-level scheduling, cross-operator synchronization, and hierarchical verification hooks.

### C.2.2 Experiment 2: Generation with the Intended Fusion Strategy

**Input to the LLM.** The LLM is explicitly told the intended fusion strategy for single-batch decode based on Llama3.2-1B. The requested fused regions include RMSNorm, QKV projection, RoPE, KV-cache update, attention, output projection, residual paths, MLP, final RMSNorm, and LM head. Unlike Experiment 1, this setting removes ambiguity about *what* should be fused.

**Models evaluated.** This experiment uses GPT-5.2, which performs best in experiment 1. It is conducted using the agent framework Codex, running on NVIDIA A100.

**Expected output.** The expected output is a correct and usable persistent MegaKernel implementation, or at least a single-entry fused device-side execution path that preserves the full decode dataflow and matches the reference implementation within the target bf16 tolerance.

**Observed output.** The initial output remained closer to a host-orchestrated multi-kernel decode pipeline than to a strict persistent MegaKernel. Through iterative engineering, the implementation could be hardened into a more MegaKernel-like structure with a single CUDA entry point and several device-side fused helper regions, including:

- RMSNorm, QKV projection, RoPE, and KV-cache update;
- output projection and residual addition;
- post-attention RMSNorm, MLP computation, and residual addition;

- final RMSNorm and LM head.

The resulting implementation could be built, imported, and executed. It also produced the same top prediction as the reference implementation in selected decode cases. However, the logits still deviated beyond the required numerical tolerance. Additional attempts to adjust fast-math behavior, RoPE handling, logits output, and terminal fusion did not reduce the error to the target level.

This result indicates that telling GPT-5.2 *what to fuse* is not sufficient. The missing capability is not merely fusion planning, but the reliable device-side realization of the fused path. In particular, the implementation must preserve cross-operator dataflow, cache layout, dtype behavior, synchronization, residual ordering, and numerical stability over a long decode chain.

#### C.2.3 Experiment 3: Adaptation from human-crafted MegaKernel Reference

**Input to the LLM.** This experiment also uses GPT-5.2. The LLM is given access to a human-expert MegaKernel implementation for Llama3.2-1B and is asked to adapt it to Qwen3-0.6B. The reference implementation contains a low-latency persistent MegaKernel path and associated runtime components.

**Models evaluated.** This experiment also uses GPT-5.2, and it is conducted using the agent framework Codex, running on NVIDIA A100, which is the same as experiment 2.

**Expected output.** The expected output is a correct MegaKernel extension for the target model. A successful adaptation should preserve the persistent execution semantics of the reference implementation while correctly adapting all model-specific details, including hidden size, number of layers, attention head structure, KV-head mapping, head dimension, intermediate size, vocabulary size, Q/K normalization, RoPE semantics, KV-cache layout, residual ordering, MLP computation, and LM head behavior. The output should run from Python and match the reference framework numerically.

**Observed output.** The LLM produced a CUDA extension that could be built and imported. An initial validation also succeeded. The generated code imitated the surface-level structure of a MegaKernel: it introduced opcode dispatch and paths corresponding to several instruction-like operations, such as fused RMSNorm/QKV computation, attention partial computation, attention reduction, output projection with residual addition, fused MLP operations, and final RMSNorm with LM head.

However, the internal implementations were simplified tensor-work approximations rather than faithful migrations of the expert MegaKernel. The generated code did not reproduce key aspects of the reference implementation, such as static scheduling and split-KV attention.

Correctness against the target model reference failed. The generated implementation produced logits that differed substantially from the framework reference and did not match the reference top prediction. Thus, even when given a human expert MegaKernel implementation, GPT-5.2 mostly reproduced the external shape of the system: extension structure, dispatch branches, opcode names, model constants, and an initial validation. It did not correctly preserve the low-level invariants and model-specific semantics required for a working MegaKernel.

### C.3 Failure Analysis

The three experiments reveal a consistent pattern. LLMs can generate plausible CUDA code and can often repair syntax- or build-level errors, but they struggle with the system-level invariants that make a MegaKernel correct.

First, direct prompting tends to fall back to the dominant patterns in public CUDA code: per-operator kernels, host-side launch sequences, and local fusion of simple adjacent operations. These patterns are useful for ordinary operator generation, but they do not provide persistent execution, instruction streams, SM-level scheduling, or cross-operator synchronization.

Second, specifying the fusion strategy does not solve the implementation problem. Knowing *what* operators should be fused is different from knowing *how* to realize the fused path on device while preserving layout, cache semantics, residual ordering, and numerical stability. Experiment 2 shows that GPT-5.2 can produce a single-entry, more MegaKernel-like implementation and still fail the logits tolerance.

Third, expert code examples are not sufficient by themselves. Human-crafted MegaKernels encode many implicit assumptions about model shape, hardware scheduling, buffer layout, page and barrier lifetimes, attention decomposition, and verification strategy. In Experiment 3, GPT-5.2 copied the visible structure of the reference implementation but failed to infer which assumptions had to be rewritten for the target model. The result was a buildable and runnable extension that failed reference correctness.

The failure modes concentrate in system integration rather than CUDA syntax alone:

- inconsistent multi-file contracts and host/device ABI mismatches;
- incorrect tensor shapes, packed layouts, or head/KV-head mappings;
- incorrect KV-cache write/read semantics;
- missing Q/K normalization or misplaced RoPE semantics;
- unstable attention softmax and reduction;
- bf16 accumulation mismatches;
- incorrect residual, RMSNorm, MLP, or LM-head ordering;
- absence of instruction-level reference execution for localizing errors.

These failures make performance benchmarking of the LLM-generated baselines inappropriate. A fast but numerically incorrect kernel is not a valid inference implementation.

### C.4 Summary

Across the three settings, we observe increasingly favorable inputs to the LLM but no successful unaided MegaKernel generation:

Table 6: Summary of LLM MegaKernel generation experiments.

| Experiment | LLM | LLM input | Observed output | Conclusion |
|---|---|---|---|---|
| Direct generation | GPT-5.2, Claude-Sonnet-4.5, DeepSeek-R1 | High-level performance goal for single-batch decode | Discrete pipelines or shallow local fusion; none passed full correctness | LLMs do not naturally generate instruction-driven persistent MegaKernels. |
| Fusion strategy provided | GPT-5.2 | Correct fused decode structure | Buildable and runnable single-entry implementation, but logits error remained above tolerance | Knowing what to fuse is insufficient without execution and verification scaffolds. |
| Expert reference provided | GPT-5.2 | Human-crafted MegaKernel reference plus target-model adaptation task | Buildable opcode-dispatch extension, but reference correctness failed | Reference code alone does not make cross-model MegaKernel migration reliable. |

Together, these results justify QIMENG-MEGAKERNEL 's instruction-centric, staged, and verifier-guided design. The core obstacle is not local CUDA code generation; it is maintaining global execution invariants across fusion planning, device-side realization, model-specific adaptation, and numerical verification.

## D Example of an Optimization Skeleton

Listing 1 shows the GEMV optimization skeleton on A100.

```
// Generic Optimization Skeleton for fused GEMV instruction in a MegaKernel
//
// Instruction abstraction:
//   Y = PostOp( GEMV( PreOp(A), W ) )
//
// Key mechanisms:
//   - activation-side pre-fusion;
//   - independent weight prefetch;
//   - global-memory counters for inter-SM instruction ordering;
//   - double-buffered GEMV pipeline;
//   - output-side post-fusion.

__device__ __forceinline__ void fused_gemv_skeleton(
  Globals& g,
  SmemLayout& smem
) {
  // 1. Parse instruction.
  // Decode layer id, output tile range, operand offsets, and global-counter
  // slots from the scheduler-generated instruction.
  ParsedInstruction inst = parse_instruction(smem.instruction);

  // 2. Prefetch the first weight tile.
  // W_tile_0 does not depend on the upstream activation value, so it can be
  // moved from global memory to shared memory before waiting on the global
  // dependency counter.
  async_load(W_tile_0, smem.W_stage[0]);
  async_fence();
  __syncthreads();

  // 3. Wait on global inter-SM dependency counter.
  // Different SMs execute different instructions. This wait enforces that
  // the activation tensor consumed by this GEMV has already been produced.
  wait_global_counter(
      g.global_counters,
      inst.wait_counter,
      inst.expected_count
  );
  __threadfence();
  __syncthreads();

  // 4. Pre-fusion on activation.
  // Example: A_pre = RMSNorm(A). The result is kept in shared memory or
  // registers for the GEMV core.
  PreOp(A, smem.A_pre);
  __syncthreads();

  // 5. Double-buffered GEMV pipeline.
  for (int tile = inst.tile_begin; tile < inst.tile_end; ++tile) {
      int curr = tile % 2;
      int next = (tile + 1) % 2;

      // Issue next weight tile.
      if (tile + 1 < inst.tile_end) {
          async_load(W_tile_next, smem.W_stage[next]);
          async_fence();
      }

      // Wait for the current weight tile before reading SMEM.
      async_wait_current();
      __syncthreads();

      // Core GEMV:
      //   acc[row] = sum_k A_pre[k] * W_tile_curr[row, k]
      GEMV(smem.A_pre, smem.W_stage[curr], smem.acc);
      __syncthreads();

      // 6. Post-fusion and store.
      // Examples: RoPE, activation, residual add, cast, packing, KV-cache
      // write, or intermediate-buffer write.
      PostOpAndStore(smem.acc, g, inst, tile);

      // 7. Publish completion through global counter.
      // The update rule is instruction-specific:
      //   - per output tile,
      //   - per attention head,
      //   - per partial block,
```

```
77        //   - or per whole instruction.
78        //
79        // The operation may be atomicAdd(delta) or another release-style update
80        // selected by the instruction contract.
81        __threadfence();
82        __syncthreads();
83        publish_global_counter(
84            g.global_counters,
85            inst.publish_counter(tile),
86            inst.publish_delta(tile)
87        );
88    }
89 }
```

Listing 1: Skeleton used by QIMENG-MEGAKERNEL to guide the generation of fused GEMV instruction on A100.

# E Examples of Generated CUDA Code

## E.1 Generated Code for rms_qkv_gemv_rope Instruction

This example shows the generated CUDA code for the single-batch `rms_qkv_gemv_rope` instruction from the Llama3.2-1B inference implementation on the NVIDIA A100 GPU. The listing was produced by Gemini-3-Pro in conjunction with QIMENG-MEGAKERNEL , guided by the GEMV optimization skeleton.

As shown in Listing 2, the implementation prefetches QKV weight tiles through a double-buffered shared-memory pipeline and, when `layer_idx > 0`, waits on a global barrier from the previous layer before consuming the hidden activation. It then applies RMSNorm to the activation vector and enters the tiled GEMV loop, where reduction along the hidden dimension is interleaved with prefetching the next weight tile. In the epilogue, RoPE is applied to the $Q$ and $K$ streams, while the $V$ stream is written directly to the KV cache.

```
 1 __device__ __forceinline__ void rms_qkv_gemv_rope(
 2     LlamaGlobals& g,
 3     SmemLayout&   smem
 4 ) {
 5     constexpr int SlicedK = 1;
 6
 7     struct ParsedInstruction {
 8         int layer_idx;
 9         int tile_M_idx;
10         int start_block_idx;
11         int end_block_idx;
12         int iters;
13         // Python serializes RMS_QKV_GEMV_RoPE as:
14         // [opcode, layer_idx, tile_M_idx, tile_N_start_idx, tile_N_end_idx,
       tile_M_granularity, tile_N_granularity]
15         // Indices:  0       1           2            3                4
       5                   6
16         __device__ __forceinline__ explicit ParsedInstruction(
17             const int32_t* smem_inst
18         ) {
19             layer_idx       = smem_inst[1];
20             tile_M_idx      = smem_inst[2];  // Not used in this kernel (always
       processing full tiles)
21             start_block_idx = smem_inst[3];  // tile_N_start_idx
22             end_block_idx   = smem_inst[4];  // tile_N_end_idx
23             iters           = end_block_idx - start_block_idx;
24         }
25     };
26
27     ParsedInstruction inst(smem.instruction);
28
29     int tid = threadIdx.x;
30
31     constexpr int TileM        = RMS_QKV_TILE_M;           // 16
32     constexpr int TileK        = RMS_QKV_TILE_K;           // 2048
33     constexpr int HeadDim      = LlamaConfig::HEAD_DIM;    // 64
34     constexpr int TilesPerHead = HeadDim / TileM;          // 4
35
36     constexpr int Q_End = LlamaConfig::HIDDEN_DIM;                              //
       2048
```

```
    constexpr int K_End = Q_End + LlamaConfig::NUM_KV_HEADS * HeadDim;           // 2560
    constexpr int V_End = K_End + LlamaConfig::NUM_KV_HEADS * HeadDim;           // 3072

    // View of the op-specific shared memory.
    auto& op_smem = *reinterpret_cast<RmsQkvRopeSmem*>(smem.main_workspace);

    // ========================================================================
    // Weight prefetch before barrier (overlap HBM with upstream spin)
    // ========================================================================
    auto t_g_qkv_w = g.get_qkv_weight()(inst.layer_idx, _, _);

    int total_tiles = inst.iters;

    if (total_tiles > 0) {
        int global_tile_idx = inst.start_block_idx;
        for (int row = 0; row < TileM; ++row) {
            const bfloat16_t* row_g = t_g_qkv_w.data().get()
                + static_cast<long>(global_tile_idx) * TileM * LlamaConfig::HIDDEN_DIM + row * LlamaConfig::HIDDEN_DIM;
            bfloat16_t* row_s = op_smem.weights.stage_0 + row * TileK;
            auto t_g_row = make_tensor(make_gmem_ptr(row_g), make_shape(Int<TileK>{}), LayoutRight{});
            auto t_s_row = make_tensor(make_smem_ptr(row_s), make_shape(Int<TileK>{}), LayoutRight{});
            load_tile_async(t_g_row, t_s_row);
        }
        cp_async_fence();
    }
    __syncthreads();

    // ========================================================================
    // Barrier Wait (prior layer DownProj complete)
    // ========================================================================
    if (inst.layer_idx > 0 && tid == 0) {
        int prev_layer  = inst.layer_idx - 1;
        int wait_opcode = OPCODE_DOWN_PROJ_RESIDUAL - 1;
        int wait_slot   = 0;

        auto t_barriers_wait = g.get_barrier();
        volatile int32_t* bar_ptr = &t_barriers_wait(prev_layer, wait_opcode, wait_slot);

        int expected_count = 512;  // Expected barrier count from DownProj
        barrier_wait(bar_ptr, expected_count);

        __threadfence();
    }
    __syncthreads();

    auto t_g_input = g.get_hidden_state();
    auto t_g_rms_w = g.get_attn_norm()(inst.layer_idx, _);
    auto t_g_q_out = g.get_q_post_rope();
    auto t_g_k_cache = g.get_k_cache()(inst.layer_idx, 0, _, _, _);
    auto t_g_v_cache = g.get_v_cache()(inst.layer_idx, 0, _, _, _);
    auto t_g_rope_cos = g.get_rope_cos()(g.pos_id, _);
    auto t_g_rope_sin = g.get_rope_sin()(g.pos_id, _);

    auto t_s_norm = make_tensor(
        make_smem_ptr(op_smem.inputs.activation),
        make_shape(Int<LlamaConfig::HIDDEN_DIM>{}),
        LayoutRight{}
    );
    auto t_reduce_workspace = make_tensor(
        make_smem_ptr(op_smem.misc.reduction_workspace),
        make_shape(Int<KernelConfig::NUM_WARPS>{}),
        LayoutRight{}
    );
    auto t_s_gemv_out = make_tensor(
        make_smem_ptr(op_smem.misc.gemv_output),
        make_shape(Int<RMS_QKV_TILE_M>{}),
        LayoutRight{}
    );

    auto t_barriers = g.get_barrier();

    // --- Fusion 1 (after G2S): RMS norm (full activation read, then normalize) ---
    rms_norm<KernelConfig::NUM_WARPS>(
        t_g_input,
        t_g_rms_w,
```

```
        t_s_norm,
        g.rms_norm_eps,
        t_reduce_workspace
    );

    __syncthreads();

    // ========================================================================
    // Main Loop (per N-tile): Issue Next -> Wait -> Compute -> Epilogue
    // ========================================================================
    for (int i = 0; i < total_tiles; ++i) {
        int stage_curr      = i % 2;
        int stage_next      = (i + 1) % 2;
        int global_tile_idx = inst.start_block_idx + i;

        // --- Phase 1: Issue Next (row-wise G2S: weight row stride = HIDDEN_DIM) ---
        if (i + 1 < total_tiles) {
            int next_global_tile_idx = inst.start_block_idx + (i + 1);
            bfloat16_t* stage_next_ptr = (stage_next == 0) ? op_smem.weights.stage_0
    : op_smem.weights.stage_1;
            for (int row = 0; row < TileM; ++row) {
                const bfloat16_t* row_g = t_g_qkv_w.data().get()
                    + static_cast<long>(next_global_tile_idx) * TileM * LlamaConfig::
    HIDDEN_DIM + row * LlamaConfig::HIDDEN_DIM;
                bfloat16_t* row_s = stage_next_ptr + row * TileK;
                auto t_g_row = make_tensor(make_gmem_ptr(row_g), make_shape(Int<TileK
    >{}), LayoutRight{});
                auto t_s_row = make_tensor(make_smem_ptr(row_s), make_shape(Int<TileK
    >{}), LayoutRight{});
                load_tile_async(t_g_row, t_s_row);
            }
            cp_async_fence();
        }

        // --- Phase 2: Wait ---
        if (i + 1 < total_tiles) {
            cp_async_wait<1>();
        } else {
            cp_async_wait<0>();
        }
        __syncthreads();

        // --- Phase 3: Compute (S2R + register GEMV + R2S) ---
        bfloat16_t* smem_ptr_curr = (stage_curr == 0)
            ? op_smem.weights.stage_0
            : op_smem.weights.stage_1;

        auto t_s_tile_curr = make_tensor(
            make_smem_ptr(smem_ptr_curr),
            make_shape(Int<TileM>{}, Int<TileK>{}),
            LayoutRight{}
        );

        compute_gemv_tile<KernelConfig::NUM_WARPS, TileM, TileK, SlicedK>(
            t_s_norm,
            t_s_tile_curr,
            t_reduce_workspace,
            t_s_gemv_out
        );
        __syncthreads();

        // --- Phase 4: Epilogue (R2S) + Fusion 2: RoPE + global write ---
        int tid_inner = threadIdx.x;
        bool row_valid = (tid_inner < TileM);
        int row_in_tile    = tid_inner;
        int global_row_idx = global_tile_idx * TileM + row_in_tile;
        int head_dim_idx   = global_row_idx % HeadDim;

        if (row_valid) {
            float val = t_s_gemv_out(row_in_tile);

            constexpr unsigned mask =
                (TileM == 32) ? 0xffffffffu : (1u << TileM) - 1;
            __syncwarp(mask);

            if (global_row_idx < Q_End) {
                float c = t_g_rope_cos(head_dim_idx);
                float s = t_g_rope_sin(head_dim_idx);
                val = apply_rope_interleaved<TileM>(
                    val, c, s, tid_inner
                );
```

```
                t_g_q_out(global_row_idx) =
                    static_cast<bfloat16_t>(val);
            }
            else if (global_row_idx < K_End) {
                float c = t_g_rope_cos(head_dim_idx);
                float s = t_g_rope_sin(head_dim_idx);
                val = apply_rope_interleaved<TileM>(
                    val, c, s, tid_inner
                );
                int k_head_idx =
                    (global_row_idx - Q_End) / HeadDim;

                t_g_k_cache(g.pos_id, k_head_idx, head_dim_idx) =
                    static_cast<bfloat16_t>(val);
            }
            else if (global_row_idx < V_End) {
                int v_head_idx =
                    (global_row_idx - K_End) / HeadDim;

                t_g_v_cache(g.pos_id, v_head_idx, head_dim_idx) =
                    static_cast<bfloat16_t>(val);
            }
        }
        __syncthreads();

        // --- [Barrier Update] ---
        __threadfence();
        __syncthreads();
        if (tid_inner == 0) {
             int head_idx = global_tile_idx / TilesPerHead;
             atomicAdd(
                 &t_barriers(inst.layer_idx,
                             OPCODE_RMS_QKV_GEMV_ROPE - 1,
                             head_idx),
                 1
             );
        }
    }
}
```

Listing 2: CUDA code generated by Gemini-3-Pro for the `rms_qkv_gemv_rope` instruction used in the Llama-3.2-1B inference implementation on A100.

### E.2 Generated Code for rms_qkv_gemm_rope Instruction

This example shows the generated CUDA code for the multi-batch `rms_qkv_gemm_rope` instruction from the same Llama3.2-1B implementation on the NVIDIA A100 GPU. The listing was also produced by Gemini-3-Pro in conjunction with QIMENG-MEGAKERNEL , but guided by the GEMM optimization skeleton.

As shown in Listing 3, the GEMM implementation first materializes the RMS-normalized activation tile as a resident shared-memory operand. It then dispatches Tensor-Core GEMM tiles over the QKV weight matrix, share the same MMA pipeline. Unlike the GEMV version, which is organized around a vector activation and CUDA-core reduction, the GEMM version relies on instruction-specific shared-memory layouts, `ldmatrix`-style shared-to-register fragment loads, and Tensor-Core MMA execution. Completion is published per attention-head tile through global counters, enabling downstream attention instructions to start once their corresponding Q/K/V tiles are ready.

```
__device__ __forceinline__ void rms_qkv_gemm_rope(
  LlamaGlobals& g,
  SmemLayout&   smem,
  int           inst_idx
) {
  // 1. Parse instruction.
  const int32_t* inst = smem.instruction;
  int layer_idx       = inst[1];
  int batch_size      = inst[2];
  int start_block_idx = inst[4];
  int end_block_idx   = inst[5];

  int tid   = threadIdx.x;
  int sm_id = blockIdx.x;

  uint8_t* smem_base = smem.main_workspace;
  float* reduce_ws = smem_ptr_at<float>(
```

```
        smem_base,
        QkvGemmSmemLayout::kOffsetReduceWorkspace
    );

    // 2. Wait for the producer of the input activation.
    // For layer > 0, QKV consumes the hidden state produced by the previous
    // layer's DownProj instruction.
    if (layer_idx > 0 && tid == 0) {
        auto barriers = g.get_barrier();

        volatile int32_t* bar_ptr =
            &barriers(layer_idx - 1, OPCODE_DOWN_PROJ_RESIDUAL - 1, 0);

        int expected = LlamaConfig::HIDDEN_DIM / KernelConfig::kTileN;

        barrier_wait(
            bar_ptr,
            expected,
            sm_id,
            OPCODE_RMS_QKV_GEMM_ROPE,
            /*check_idx=*/0,
            layer_idx,
            &g,
            inst_idx
        );

        __threadfence();
    }
    __syncthreads();

    // 3. Pre-operation: RMSNorm writes the resident A operand into shared memory.
    // The normalized activation tile is reused by all QKV GEMM N-tiles.
    const bfloat16_t* norm_w =
        g.attn_norm_ptr + (long)layer_idx * LlamaConfig::HIDDEN_DIM;

    rms_norm_batched<
        KernelConfig::NUM_THREADS,
        LlamaConfig::HIDDEN_DIM,
        GemmConfig1Tile::SmemLayoutA
    >(
        g.hiddenem_base,
            QkvGemmSmem4. Dispatch tiled QKV GEMM.
    // The B operand is the QKV weight. Adjacent N-tiles are grouped
    // into 4/2/1-tile variants to reuse the same GEMM pipeline structure while
    // increasing Tensor Core work per instruction when possible.
    const bfloat16_t* qkv_w =
        g.qkv_weight_ptr
        + (long)layer_idx
        * LlamaConfig::QKV_OUT_DIM
        * LlamaConfig::HIDDEN_DIM;

    auto barriers = g.get_barrier();
    constexpr int TilesPerHead =
        LlamaConfig::HEAD_DIM / KernelConfig::kTileN;

    int i = start_block_idx;
    while (i < end_block_idx) {
        bool use_four_tiles = (i + 3) < end_block_idx;
        bool use_two_tiles  = !use_four_tiles && (i + 1) < end_block_idx;

        int n_blocks =
            use_four_tiles ? 4 :
            use_two_tiles  ? 2 :
                             1;

        const bfloat16_t* gmem_B =
            qkv_w + (long)i
            * KernelConfig::kTileN
            * LlamaConfig::HIDDEN_DIM;

        int tile_n_global = i * KernelConfig::kTileN;

        if (use_four_tiles) {
            compute_gemm_tile<GemmConfig4Tile>(
                tid,
                batch_size,
                smem_base,
                gmem_B,
                tile_n_global,
                layer_idx,
                g,
```

```
                sm_id,
                inst_idx
            );
        } else if (use_two_tiles) {
            compute_gemm_tile<GemmConfig2Tile>(
                tid,
                batch_size,
                smem_base,
                gmem_B,
                tile_n_global,
                layer_idx,
                g,
                sm_id,
                inst_idx
            );
        } else {
            compute_gemm_tile<GemmConfig1Tile>(
                tid,
                batch_size,
                smem_base,
                gmem_B,
                tile_n_global,
                layer_idx,
                g,
                sm_id,
                inst_idx
            );
        }

        // 5. Publish completion.
        // Downstream attention instructions wait per attention head. When a
        // grouped GEMM spans multiple logical N-tiles, the counter update is
        // coalesced per head.
        __threadfence();
        __syncthreads();

        if (tid == 0) {
            int tile_offset = 0;

            while (tile_offset < n_blocks) {
                int head_idx = (i + tile_offset) / TilesPerHead;

                int tiles_in_head = 1;
                while (
                    tile_offset + tiles_in_head < n_blocks &&
                    (i + tile_offset + tiles_in_head) / TilesPerHead == head_idx
                ) {
                    ++tiles_in_head;
                }

                atomicAdd(
                    &barriers(
                        layer_idx,
                        OPCODE_RMS_QKV_GEMM_ROPE - 1,
                        head_idx
                    ),
                    tiles_in_head
                );

                tile_offset += tiles_in_head;
            }
        }

        i += n_blocks;
    }
}
```

Listing 3: CUDA code generated by Gemini-3-Pro for the `rms_qkv_gemm_rope` instruction used in the Llama-3.2-1B inference implementation on A100. Shared-memory layout definitions, MMA configuration, `ldmatrix`-style fragment-load setup, and other low-level helper routines are omitted for brevity.

# F Overall Workflow of QiMeng-MegaKernel

This section summarizes the overall workflow of QiMeng-MegaKernel. Given a Hugging Face model, QiMeng-MegaKernel decomposes the model into instruction instances, analyzes both algorithm-related and GPU-related configurations, schedules the instructions, generates Python ref-

erences and CUDA implementations, and finally assembles the verified device functions into a single MegaKernel. The whole process is performed in an iterative generate–verify–refine loop.

**Algorithm 1** Overall workflow of QIMENG-MEGAKERNEL

**Require:** Hugging Face model $\mathcal{M}$, optimization skeletons $\mathcal{S}$, target GPU specification $\mathcal{G}$, tolerance $\epsilon$, maximum refinement rounds $T$
**Ensure:** MegaKernel assembled from verified CUDA functions, or generation failure

$\mathcal{F} \leftarrow \emptyset$
**for** $t = 1, \ldots, T$ **do**
  **Decomposition and Configuration**
  $\mathcal{I} \leftarrow \textsc{DecomposeModel}(\mathcal{M}, \mathcal{F})$
  **for** each instruction instance $I_i \in \mathcal{I}$ **do**
    $(C_i^{\mathrm{alg}}, C_i^{\mathrm{gpu}}) \leftarrow \textsc{AnalyzeConfig}(I_i, \mathcal{M}, \mathcal{G}, \mathcal{F})$
  **end for**
  **Instruction Scheduling**
  $\pi \leftarrow \textsc{ScheduleInstructions}(\mathcal{I}, \{C_i^{\mathrm{alg}}\})$
  $\mathcal{L} \leftarrow \textsc{MapInstructionsToSMs}(\pi, \{C_i^{\mathrm{gpu}}\}, \mathcal{G})$
  **Python Reference Generation**
  **for** each instruction instance $I_i \in \mathcal{I}$ **do**
    $P_i \leftarrow \textsc{GeneratePythonReference}(I_i, C_i^{\mathrm{alg}}, \mathcal{F})$
  **end for**
  **Stage 1: HF–Python Verification**
  $(b_1, \mathcal{F}_1) \leftarrow \textsc{VerifyHFvsPython}(\mathcal{M}, \{P_i\}, \pi, \epsilon)$
  **if** $b_1 = \text{false}$ **then**
    $\mathcal{F} \leftarrow \mathcal{F}_1$; **continue** to the next refinement round
  **end if**
  **CUDA Generation**
  $\mathcal{D} \leftarrow \emptyset$
  **for** each instruction instance $I_i \in \mathcal{I}$ **do**
    $S_i \leftarrow \mathcal{S}[\textsc{Type}(I_i)]$
    $D_i \leftarrow \textsc{GenerateCUDADeviceFunction}(I_i, S_i, P_i, C_i^{\mathrm{gpu}}, \mathcal{F})$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{D_i\}$
  **end for**
  **Stage 2: Python–CUDA Verification**
  $b_2 \leftarrow \text{true}$; $\mathcal{F}_2 \leftarrow \emptyset$
  **for** each instruction instance $I_i \in \mathcal{I}$ **do**
    $(ok_i, f_i) \leftarrow \textsc{VerifyCUDAvsPython}(D_i, P_i, \epsilon)$
    **if** $ok_i = \text{false}$ **then**
      $b_2 \leftarrow \text{false}$; $\mathcal{F}_2 \leftarrow \mathcal{F}_2 \cup \{f_i\}$
    **end if**
  **end for**
  **if** $b_2 = \text{false}$ **then**
    $\mathcal{F} \leftarrow \mathcal{F}_2$; **continue** to the next refinement round
  **end if**
  **MegaKernel Assembly**
  $\mathcal{K}^* \leftarrow \textsc{AssembleMegaKernel}(\mathcal{D}, \pi, \mathcal{L}, \mathcal{G})$
  **return** $\mathcal{K}^*$
**end for**
Report generation failure

## G Generation Cost of MegaKernel

The generation of a single MegaKernel consists of two levels: the strategy level and the implementation level. The strategy level includes fusion instantiation, scheduling, mapping, and Python-based verification, and typically requires about 5M tokens across roughly 20 iterations, taking approximately 1–2 hours. Under GPT-5.2-Codex pricing, this stage costs about 15 USD. The implementation level includes CUDA code generation and tuning, and typically requires about 30M tokens across roughly 20 iterations, taking approximately 4–5 hours, with an estimated cost of about 90 USD. In total, generating a single MegaKernel requires approximately 35M tokens and 5–7 hours.

## H Detailed Latency Analysis

This section complements the theoretical latency analysis in Section 3.6. While the main text summarizes the latency saving of QIMENG-MEGAKERNEL using four terms, we further explain the source of each term and its relation to workload scale. For readability, Section 3.6 writes the latency terms as functions of batch size $B$ and context length $L$. Among these factors, $B$ affects all types of instructions. In contrast, $L$ mainly affects attention instructions, while its impact on other instructions is weaker. The last term captures the extra overhead introduced by the generated MegaKernel, including higher register pressure, larger shared-memory usage, and possible occupancy reduction, which can become more significant for larger workloads.

**Launch overhead.** The first term is the launch saving

$$S_{\text{launch}} = (K-1) \times \bar{t}_{\text{launch}}. \tag{1}$$

In discrete-kernel execution, one decoding step invokes $K$ kernels, while QIMENG-MEGAKERNEL uses a single generated MegaKernel. Therefore, $K-1$ kernel launches are removed. The term $\bar{t}_{\text{launch}}$ is the average launch overhead of one GPU kernel under a fixed hardware and runtime environment.

Since $S_{\text{launch}}$ is mainly determined by $K$ and $\bar{t}_{\text{launch}}$, it is a stable absolute saving. However, when normalized by batch size, the saving becomes

$$\frac{S_{\text{launch}}}{B} = \frac{(K-1) \times \bar{t}_{\text{launch}}}{B}. \tag{2}$$

Thus, as $B$ grows, the launch saving per batch is amortized. Similarly, as the workload scale increases through larger $B$, longer attention context $L$, or a larger model, useful computation occupies a larger fraction of the decoding step, so the relative contribution of launch saving decreases.

**Memory accessing.** We compare the cost of executing $K$ discrete kernels with the cost of executing $K'$ fused instruction segments inside the generated MegaKernel. Consistent with Section 3.6, we decompose the fusion saving into a device-memory access term and a remaining execution term:

$$S_{\text{fusion}}(B, L) = S_{\text{mem}}(B, L) + S_{\text{exec}}(B, L), \tag{3}$$

where

$$S_{\text{mem}}(B, L) = \sum_{i=1}^{K} T^{i}_{\text{dk_mem}}(B, L) - T_{\text{mk_mem}}(B, L), \tag{4}$$

and

$$S_{\text{exec}}(B, L) = \sum_{i=1}^{K} T^{i}_{\text{dk_exec}}(B, L) - \sum_{j=1}^{K'} T^{j}_{\text{mk_exec}}(B, L). \tag{5}$$

The memory-related term $S_{\text{mem}}(B, L)$ captures the reduction in exposed GPU-side memory-access costs. In discrete-kernel execution, intermediate results produced by one kernel are materialized in device global memory and accessed by subsequent kernels. Instruction-level fusion allows intermediate results to remain in registers or shared memory whenever possible, reducing redundant accesses to global memory. Software pipelining can also overlap memory accesses with computation, reducing the memory latency exposed on the critical path. These benefits depend on the fusion pattern and available on-chip storage; a single persistent kernel does not by itself eliminate all global-memory accesses between instructions.

Here, $T^{i}_{\text{dk_mem}}(B, L)$ and $T_{\text{mk_mem}}(B, L)$ represent exposed device-memory access costs in the discrete and fused executions, respectively. The execution terms represent the remaining device-side execution costs, excluding the memory costs already accounted for in $S_{\text{mem}}(B, L)$. This decomposition avoids counting overlapped computation and memory accesses twice. Kernel launch costs are accounted for separately in $S_{\text{launch}}$, and SM-level imbalance is addressed by the execution-bubble term below.

The execution-related saving $S_{\text{exec}}(B, L)$ does not require reducing the mathematical computation of the model. Instead, fusion can reduce repeated auxiliary operations and execution bookkeeping across discrete kernels. Its benefit depends on the generated implementations and the workload.

For small $B$ and $L$, low arithmetic intensity makes exposed memory-access costs a larger fraction of latency, so on-chip reuse and reduced intermediate materialization can provide substantial relative savings. As the workload grows, the absolute memory saving need not decrease: the volume of intermediate data can also increase. However, increased useful computation can reduce its relative contribution. Longer contexts additionally increase attention and KV-cache traffic, which cannot all be eliminated by instruction fusion. These effects explain the workload-dependent fusion saving described in the main text.

**Execution bubbles.** The third term $T_{\text{bubble}}(B, L)$ denotes the execution bubbles reduced by instruction scheduling and SM-level mapping. These bubbles mainly come from tail latency across SMs: when different SMs receive uneven workloads, the faster SMs wait for the slowest SM to finish. For an instruction layout $\pi$, let $W_s^\pi(B, L)$ be the workload assigned to SM $s$, and let $S$ be the number of SMs. The stage completion time is dominated by the slowest SM:

$$T_{\text{stage}}^\pi(B, L) = \max_s W_s^\pi(B, L). \tag{6}$$

The tail bubble under layout $\pi$ can be approximated as the gap between the slowest SM and the average SM workload:

$$T_{\text{tail}}^\pi(B, L) \approx \max_s W_s^\pi(B, L) - \frac{1}{S}\sum_{s=1}^{S} W_s^\pi(B, L). \tag{7}$$

We use $T_{\text{tail}}^{\text{base}}(B, L)$ to denote the tail bubble under a baseline layout, and $T_{\text{tail}}^{\text{AMK}}(B, L)$ to denote the tail bubble under the layout generated by QIMENG-MEGAKERNEL . QIMENG-MEGAKERNEL reduces this tail bubble by mapping instruction instances across SMs according to their estimated execution costs. The generated layout makes the workloads of different SMs more balanced, so

$$T_{\text{tail}}^{\text{AMK}}(B, L) < T_{\text{tail}}^{\text{base}}(B, L). \tag{8}$$

Therefore, the reduced execution bubble is

$$T_{\text{bubble}}(B, L) = T_{\text{tail}}^{\text{base}}(B, L) - T_{\text{tail}}^{\text{AMK}}(B, L). \tag{9}$$

The relative contribution of the third term can be written as

$$\rho_{\text{bubble}}(B, L) = \frac{T_{\text{bubble}}(B, L)}{T_{\text{stage}}^{\text{base}}(B, L)}. \tag{10}$$

Since $T_{\text{bubble}}(B, L)$ is the reduced tail latency and $T_{\text{stage}}^{\text{base}}(B, L)$ grows with useful computation as $B$ or $L$ increases, $\rho_{\text{bubble}}(B, L)$ becomes smaller. Thus, $T_{\text{bubble}}(B, L)$ contributes more when tail latency is a large fraction of the stage time, and its relative effect decreases as useful computation grows.

**MegaKernel overhead.** The last term $T_{\text{overhead}}(B, L)$ captures the extra cost introduced by the generated MegaKernel. Following the main text, we model it as

$$T_{\text{overhead}}(B, L) \approx T_{\text{reg}}(B, L) + T_{\text{smem}}(B, L) + T_{\text{occ}}(B, L), \tag{11}$$

where $T_{\text{reg}}(B, L)$ denotes the overhead caused by higher register pressure, $T_{\text{smem}}(B, L)$ denotes the overhead related to shared-memory usage, and $T_{\text{occ}}(B, L)$ denotes the performance loss caused by possible occupancy reduction.

Register pressure is the most direct source of this overhead. Since a MegaKernel executes multiple fused instructions within one kernel, the generated code needs to maintain more local variables, pointers, metadata, accumulators, and intermediate values at the same time. This can increase register usage and may lead to register spilling, which introduces additional local-memory accesses and reduces execution efficiency.

Shared memory is used to explicitly manage on-chip residency in the generated MegaKernel, but it is still a limited SM resource. Larger shared-memory usage, together with higher register usage, can reduce the number of resident blocks or warps on an SM, which is captured by $T_{\text{occ}}(B, L)$.

As $B$ and $L$ increase, the positive savings from launch elimination, memory accessing reduction, and bubble reduction are amortized by more useful computation. In contrast, the register and shared-memory pressure of the MegaKernel is not amortized in the same way. Therefore, $T_{\text{overhead}}(B, L)$ becomes more significant for larger workloads and weakens the latency saving.

# I Vanilla Prompt for Ablation Studies

Listing 4 presents the vanilla prompt used in our experimental setup to decompose the MegaKernel deployment task for base LLM ablation.

```
I will provide you with the Transformers inference code of a certain model.
Without invoking any external inference frameworks, you are required to:
1.  Load the xxx model directly.
2.  Manually define the input prompt.
3.  Explicitly read the model weights and design your own inference logic.
4.  Implement and optimize the inference pipeline.
5.  Produce the correct conversational output.
6.  Finally, report the decode-stage throughput in tokens per second (tokens/s).

Model path: ""

# Transformers inference code (to be provided)
```

Listing 4: Vanilla prompt used to decompose the MegaKernel deployment task in base LLM ablation.

# J Raw Measurement Data for Overall Performance

Figures 4 and 5 report the overall performance of QIMENG-MEGAKERNEL compared to all baselines across varying output sequence lengths and a wide range of real-world evaluations.

For clarity, The figures in the main text report aggregated throughput metrics, while this appendix provides the corresponding raw throughput measurements (tokens/s) used to generate Figures 4 and 5.

Table 7 reports throughput under varying output sequence lengths on an H100 GPU with the batch size fixed to 1, corresponding to Fig. 4(a). Table 8 presents throughput across different benchmark datasets on an H100 GPU with batch size 1, corresponding to Fig. 4(b); Tables 9 and 10 provide the raw measurements underlying Fig. 5(a–d), which evaluates cross-GPU performance for the Llama3.2-1B model under matched experimental settings. In particular, Table 9 reports throughput under varying output sequence lengths on A100 and L20 GPUs with batch size 1, while Table 10 reports throughput across different benchmark datasets on the same GPUs; Table 11 reports throughput under varying batch sizes and output sequence lengths for Llama-3.2-1B on the H100 GPU, corresponding to Fig. 5(e–g). All the missing results are denoted by "–".

Table 7: Throughput (tokens/s) under different sequence lengths on H100 GPU. Batch size is fixed to 1.

| Model | Framework | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|---|
| Llama3.2-1B | Ollama | 214.12 | 270.81 | 286.44 | 288.95 |
| | vLLM | 420.61 | 431.55 | 434.03 | 436.95 |
| | TensorRT-LLM | 527.67 | 538.48 | 544.56 | 544.58 |
| | SGLang | 567.34 | 572.33 | 575.52 | 574.37 |
| | MPK | 764.58 | 716.95 | 676.13 | 630.40 |
| | Spector et al. | 925.22 | 892.91 | 837.90 | 747.22 |
| | **Ours** | **936.44** | **918.12** | **887.12** | **834.00** |
| Llama3.2-3B | Ollama | 155.77 | 185.92 | 199.05 | 195.48 |
| | vLLM | 233.50 | 233.91 | 233.99 | 236.86 |
| | TensorRT-LLM | 239.82 | 241.67 | 241.41 | 241.00 |
| | SGLang | 276.08 | 277.15 | 277.60 | 275.80 |
| | MPK | 332.18 | 311.79 | 291.17 | 236.27 |
| | Spector et al. | – | – | – | – |
| | **Ours** | **362.09** | **353.14** | **333.82** | **317.78** |
| Qwen2.5-3B | Ollama | 121.95 | 150.65 | 166.33 | 172.22 |
| | vLLM | 213.81 | 213.74 | 215.08 | 221.14 |
| | TensorRT-LLM | 241.62 | 245.17 | 247.12 | 248.08 |
| | SGLang | 241.16 | 242.44 | 253.04 | 253.86 |
| | MPK | – | – | – | – |
| | Spector et al. | – | – | – | – |
| | **Ours** | **363.74** | **355.66** | **338.54** | **325.03** |
| Qwen1.5-MoE-A2.7B | Ollama | – | – | – | – |
| | vLLM | 217.85 | 217.26 | 219.63 | 224.94 |
| | TensorRT-LLM | 232.34 | 235.66 | 238.62 | 239.42 |
| | SGLang | 284.81 | 292.25 | 296.49 | 295.78 |
| | MPK | – | – | – | – |
| | Spector et al. | – | – | – | – |
| | **Ours** | **435.93** | **434.74** | **414.30** | **405.04** |
| Qwen3-0.6B | Ollama | 248.58 | 296.97 | 329.87 | 334.25 |
| | vLLM | 358.96 | 351.97 | 370.58 | 364.17 |
| | TensorRT-LLM | 490.84 | 503.51 | 509.92 | 512.78 |
| | SGLang | 516.46 | 516.82 | 521.51 | 520.73 |
| | MPK | 714.29 | 669.12 | 616.34 | 557.69 |
| | Spector et al. | – | – | – | – |
| | **Ours** | **1146.07** | **1123.44** | **1084.58** | **1013.27** |

Table 8: Throughput (tokens/s) across different benchmarks on H100 GPU. Batch size is fixed to 1.

| Model | Framework | HumanEval | GSM8K | CoinFlip | MultiArith | ReClor | SQuAD |
|---|---|---|---|---|---|---|---|
| Llama3.2-1B | Ollama | 292.12 | 254.27 | 210.36 | 221.53 | 182.68 | 188.93 |
| | vLLM | 428.56 | 428.13 | 411.59 | 410.05 | 321.31 | 308.38 |
| | TensorRT-LLM | 538.35 | 535.40 | 536.52 | 512.54 | 532.31 | 516.06 |
| | SGLang | 561.29 | 568.47 | 489.69 | 471.54 | 560.61 | 529.04 |
| | MPK | 600.77 | 591.78 | 586.00 | 590.53 | 616.05 | 606.40 |
| | Spector et al. | 773.33 | 790.30 | 811.31 | 803.87 | 806.89 | 819.44 |
| | **Ours** | **854.19** | **872.27** | **884.29** | **887.13** | **855.22** | **878.47** |
| Llama3.2-3B | Ollama | 194.96 | 193.63 | 172.73 | 172.39 | 132.56 | 90.37 |
| | vLLM | 234.75 | 232.73 | 221.94 | 228.85 | 205.68 | 190.67 |
| | TensorRT-LLM | 240.12 | 240.98 | 239.99 | 237.02 | 241.27 | 239.78 |
| | SGLang | 269.14 | 270.61 | 268.83 | 265.31 | 271.65 | 270.48 |
| | MPK | 273.85 | 273.97 | 267.37 | 273.73 | 260.59 | 274.07 |
| | Spector et al. | – | – | – | – | – | – |
| | **Ours** | **322.77** | **335.26** | **330.75** | **333.44** | **335.05** | **332.84** |
| Qwen2.5-3B | Ollama | 184.41 | 185.53 | 166.00 | 163.13 | 143.82 | 151.67 |
| | vLLM | 219.51 | 218.58 | 218.04 | 215.14 | 171.82 | 167.58 |
| | TensorRT-LLM | 248.15 | 248.71 | 248.66 | 248.64 | 248.79 | 248.64 |
| | SGLang | 250.80 | 254.46 | 251.29 | 244.93 | 251.56 | 248.90 |
| | MPK | – | – | – | – | – | – |
| | Spector et al. | – | – | – | – | – | – |
| | **Ours** | **330.58** | **323.64** | **348.52** | **349.51** | **308.13** | **315.05** |
| Qwen1.5-MoE-A2.7B | Ollama | – | – | – | – | – | – |
| | vLLM | 219.63 | 216.21 | 171.70 | 207.18 | 186.38 | 176.96 |
| | TensorRT-LLM | 229.03 | 223.52 | 207.16 | 209.43 | 225.45 | 213.87 |
| | SGLang | 282.49 | 275.76 | 223.11 | 221.75 | 269.12 | 240.44 |
| | MPK | – | – | – | – | – | – |
| | Spector et al. | – | – | – | – | – | – |
| | **Ours** | **441.19** | **439.04** | **444.12** | **439.78** | **441.78** | **443.49** |
| Qwen3-0.6B | Ollama | 314.94 | 338.40 | 317.56 | 317.80 | 304.84 | 297.36 |
| | vLLM | 377.22 | 342.69 | 338.65 | 348.81 | 316.61 | 223.26 |
| | TensorRT-LLM | 509.34 | 511.23 | 511.66 | 512.06 | 512.69 | 512.09 |
| | SGLang | 516.32 | 519.10 | 475.83 | 462.50 | 480.26 | 517.51 |
| | MPK | 573.69 | 581.05 | 584.00 | 585.40 | 590.34 | 591.24 |
| | Spector et al. | – | – | – | – | – | – |
| | **Ours** | **1100.36** | **1118.26** | **1119.84** | **1122.77** | **1125.56** | **1127.11** |

Table 9: Throughput (tokens/s) under different sequence lengths for Llama3.2-1B on A100 and L20 GPUs. Batch size is fixed to 1.

| GPU | Framework | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|---|
| A100 | Ollama | 113.27 | 118.92 | 127.98 | 139.44 |
| | vLLM | 283.01 | 281.17 | 285.90 | 286.30 |
| | TensorRT-LLM | 302.50 | 302.12 | 261.67 | 283.24 |
| | SGLang | 290.32 | 323.60 | 341.09 | 349.76 |
| | MPK | 366.42 | 382.32 | 382.85 | 372.44 |
| | **Ours** | **458.82** | **458.66** | **456.93** | **456.69** |
| L20 | Ollama | 261.74 | 284.30 | 295.78 | 304.88 |
| | vLLM | 222.03 | 222.44 | 223.57 | 223.48 |
| | TensorRT-LLM | – | – | – | – |
| | SGLang | 252.15 | 253.70 | 254.10 | 254.39 |
| | MPK | – | – | – | – |
| | **Ours** | **403.26** | **402.25** | **400.48** | **396.76** |

Table 10: Throughput (tokens/s) across different benchmarks for Llama3.2-1B on A100 and L20 GPUs. Batch size is fixed to 1.

| GPU | Framework | HumanEval | GSM8K | CoinFlip | MultiArith | ReClor | SQuAD |
|---|---|---|---|---|---|---|---|
| A100 | Ollama | 152.95 | 126.74 | 105.86 | 103.20 | 84.56 | 95.18 |
| | vLLM | 287.81 | 286.47 | 286.26 | 275.97 | 286.31 | 283.33 |
| | TensorRT-LLM | 286.31 | 308.01 | 282.02 | 307.07 | 277.43 | 307.91 |
| | SGLang | 357.95 | 353.15 | 360.92 | 342.93 | 359.58 | 348.33 |
| | MPK | 361.78 | 355.70 | 335.22 | 354.29 | 354.50 | 360.18 |
| | **Ours** | **458.49** | **460.31** | **460.95** | **463.06** | **459.60** | **463.47** |
| L20 | Ollama | 265.38 | 275.38 | 247.11 | 234.28 | 230.61 | 173.49 |
| | vLLM | 223.39 | 222.53 | 223.10 | 223.19 | 219.71 | 215.20 |
| | TensorRT-LLM | – | – | – | – | – | – |
| | SGLang | 253.06 | 253.24 | 257.11 | 253.09 | 240.47 | 230.75 |
| | MPK | – | – | – | – | – | – |
| | **Ours** | **407.32** | **408.98** | **409.03** | **412.11** | **408.16** | **412.54** |

Table 11: Throughput (tokens/s) under different batch sizes and sequence lengths on H100 GPU for Llama3.2-1B. Missing results are denoted by "–".

| Batch | Framework | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|---|
| 2 | Ollama | – | – | – | – |
| | vLLM | 715.24 | 730.12 | 729.08 | 741.12 |
| | TensorRT-LLM | 905.86 | 933.22 | 983.33 | 1018.90 |
| | SGLang | 1038.54 | 1057.28 | 1108.16 | 1081.76 |
| | **Ours** | **1295.10** | **1285.68** | **1275.68** | **1227.00** |
| 4 | Ollama | – | – | – | – |
| | vLLM | 1402.52 | 1381.72 | 1422.44 | 1430.56 |
| | TensorRT-LLM | 1873.44 | 1985.70 | 2042.85 | 2057.21 |
| | SGLang | 2010.28 | 2099.40 | 2095.48 | 2091.16 |
| | **Ours** | **2429.32** | **2412.32** | **2351.12** | **2195.56** |
| 8 | Ollama | – | – | – | – |
| | vLLM | 2706.16 | 2820.16 | 2843.52 | 2831.28 |
| | TensorRT-LLM | 3764.23 | 3956.54 | 4050.90 | 4053.28 |
| | SGLang | 3914.32 | 3963.84 | 4088.48 | **4116.88** |
| | **Ours** | **4454.32** | **4408.96** | **4163.84** | 3625.92 |

## NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The claims in the abstract and introduction are consistent with the experimental results presented in Section 4 and Appendix J.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: The limitation is discussed in Section 6.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [Yes]

Justification: Section 3.6 provides the latency analysis and states the main assumptions. Appendix H gives the detailed derivation, showing that QIMENG-MEGAKERNEL is most advantageous for small-batch and short-to-medium-context decoding.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: In the appendix, we provide the pseudocode for the entire workflow, along with descriptions of the inputs and outputs for each step. We also present examples of key optimization skeletons and the correspondingly generated CUDA code to demonstrate the reproducibility of our method.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in

some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: The code is available at https://github.com/0o0o0o0osky/QiMeng-MegaKernel.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (https://neurips.cc/public/guides/CodeSubmissionPolicy) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (https://neurips.cc/public/guides/CodeSubmissionPolicy) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: The experimental setting and details are specified in Section 4.1.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: As stated in Section 4.1, all throughput and latency results are measured using a fixed evaluation protocol and averaged over 10 measured runs after 5 warmup runs.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.

- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: The information is provided in Section 4.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics https://neurips.cc/public/EthicsGuidelines?

   Answer: [Yes]

   Justification: The research presented in this paper complies with the NeurIPS Code of Ethics.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [N/A]

    Justification: There is no societal impact of the work performed.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.

- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The paper poses no such risks.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [N/A]

    Justification: The paper does not use existing assets.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.

- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, paperswithcode.com/datasets has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [N/A]

    Justification: The paper does not release new assets.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.

- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: LLMs are a core component of the proposed framework. Their usage is described in Section 3.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.