MOBILEKERNELBENCH: CAN LLMS WRITE EFFICIENT KERNELS FOR MOBILE DEVICES?

Xingze Zou [1 *] **Jing Wang** [1 *] **Yuhua Zheng**[1] **Xueyi Chen**[2] **Haolei Bai**[2] **Lingcheng Kong**[3]
Syed A.R. Abu-Bakar[4] **Zhaode Wang**[5] **Chengfei Lv**[5] **Haoji Hu**[1 †] **Huan Wang**[2 †]

[1]Zhejiang University [2]Westlake University [3]HKUST [4]Universiti Teknologi Malaysia [5]Alibaba

{zeezou, j_wang, YuhuaZheng, haoji_hu}@zju.edu.cn
{chenxueyi, wanghuan}@westlake.edu.cn, Baih0011@e.ntu.edu.sg
LingchengKong05@outlook.com, e-syed@utm.my
{zhaode.wzd, chengfei.lcf}@alibaba-inc.com

ABSTRACT

Large language models (LLMs) have demonstrated remarkable capabilities in code generation, yet their potential for generating kernels specifically for mobile devices remains largely unexplored. In this work, we extend the scope of automated kernel generation to the mobile domain to investigate the central question: **Can LLMs write efficient kernels for mobile devices?** To enable systematic investigation, we introduce **MobileKernelBench**, a comprehensive evaluation framework comprising a benchmark prioritizing operator diversity and cross-framework interoperability, coupled with an automated pipeline that bridges the host-device gap for on-device verification. Leveraging this framework, we conduct extensive evaluation on the CPU backend of Mobile Neural Network (MNN), revealing that current LLMs struggle with the engineering complexity and data scarcity inherent to mobile frameworks; standard models and even fine-tuned variants exhibit high compilation failure rates (over 54%) and negligible performance gains due to hallucinations and a lack of domain-specific grounding. To overcome these limitations, we propose the Mobile Kernel Agent (**MoKA**), a multi-agent system equipped with repository-aware reasoning and a plan-and-execute paradigm. Validated on MobileKernelBench, MoKA achieves state-of-the-art performance, boosting compilation success to 93.7% and enabling 27.4% of generated kernels to deliver measurable speedups over native libraries.

1 INTRODUCTION

Large language models (LLMs) have achieved remarkable success in the field of code generation, with performance further enhanced by specialized methodologies such as domain-adaptive fine-tuning and agentic reasoning frameworks (OpenAI, 2025; Anthropic, 2025; Google DeepMind, 2025; Yang et al., 2025; Team et al., 2025; Zeng et al., 2025). Building on this foundation, a series of recent studies have begun to systematically evaluate the application of LLMs in synthesizing performance-optimized CUDA kernels (Ouyang et al., 2025; Wen et al., 2025; Li et al., 2025a). These works demonstrate that LLMs can exhibit a significant degree of hardware awareness, enabling them to write high-performance kernels tailored for server-grade GPUs. With the rapid surge in demand for mobile AI applications, on-device inference has attracted increasing attention for its data safety, low inference latency, and personalized service. However, deploying deep learning models on mobile devices also demands substantial effort in kernel development, constituting a highly challenging engineering barrier that has not yet been investigated. In this work, we extend the scope of automated kernel generation to the mobile domain, presenting a preliminary investigation into the following question: *Can LLMs write efficient kernels for mobile devices?*

[*] Equal Contribution
[†] Corresponding Author

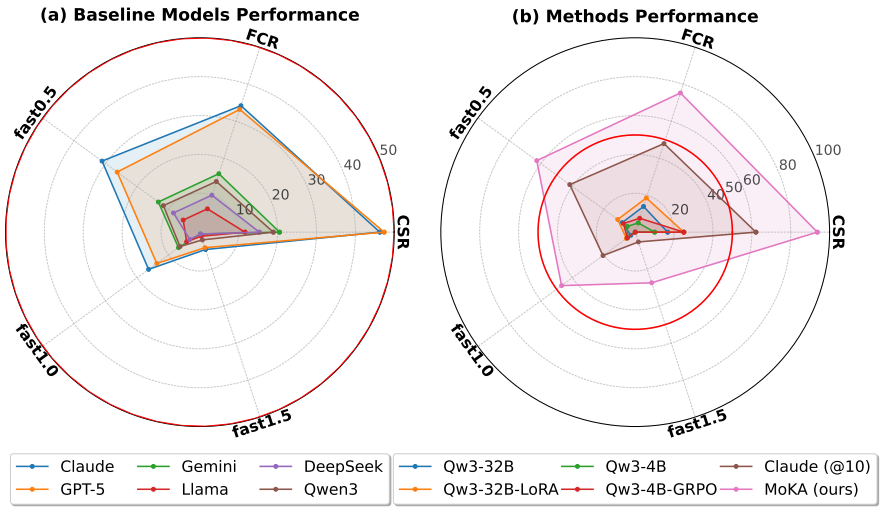


Figure 1: **Performance evaluation on MobileKernelBench across three metrics:** compilation success rate (CSR), functional correctness rate (FCR), and performance speedup (fast_p) (Ouyang et al., 2025). **(a) Baseline LLM performance:** We benchmark prevalent open- and closed-source LLMs, revealing significant shortcomings in their ability to generate functional and efficient mobile kernels. **(b) Method comparison**: We compare our proposed MoKA against common training methods, including LoRA and GRPO. The red circle (marked at 50%) corresponds to the outer limit of plot (a), highlighting that MoKA achieves substantial improvements, surpassing the performance ceiling of both baseline models and naive fine-tuning approaches.

We first investigate and compare the differences between server-side and mobile-side kernel development in Tab. 1. Mobile kernel development functions as the critical bridge for migrating models from training environments to resource-constrained edge devices, characterized by three distinct features: (1) **Compatibility priority**: the primary objective is to ensure broad support, necessitating the implementation of a vast spectrum of operators to cover diverse training frameworks; (2) **Engineering complexity**: the development process, however, is severely hampered by the fragmentation of the mobile ecosystem, where developers must navigate a myriad of heterogeneous backends and architectures; and (3) **Data scarcity:** the nascent nature of the mobile inference landscape results in a lack of high-quality reference implementations, creating a data-poor environment that poses significant generalization challenges for LLMs. Consequently, existing research is ill-suited for this specific domain: current benchmarks prioritize the algorithmic complexity of kernels rather than the operator diversity required for edge compatibility, while the tight coupling between kernels and the corresponding framework has precluded the establishment of systematic pipelines for evaluating LLM-generated kernels directly on mobile devices.

To address these limitations, we introduce **MobileKernelBench**, as illustrated in Fig. 2, a comprehensive system comprising a dedicated benchmark and an automated evaluation pipeline tailored for mobile frameworks. Diverging from the difficulty-tiered classification of prior works (Ouyang et al., 2025; Wen et al., 2025), our benchmark prioritizes operator diversity by curating 190 tasks across 12 categories of 95 primitive operators to facilitate wider model migration. Simultaneously, we ensure cross-framework interoperability via standardized PyTorch and Open Neural Network Exchange (ONNX) test data, acting as a universal bridge to mitigate inconsistent framework support and tackle ecosystem fragmentation. Complementing the benchmark, we introduce an automated evaluation pipeline to bridge the separation between host-side development and device-side testing inherent to mobile deployment. By automating the entire lifecycle spanning registration, cross-compilation, and on-device verification, it not only satisfies the concerns mentioned in previous work (Ouyang et al., 2025) but also streamlines the desktop-to-mobile workflow to capture granular debugging and performance data akin to a real-world developer's environment.

Building on the proposed evaluation pipeline design, we establish a concrete evaluation environment using Mobile Neural Network (MNN) (Jiang et al., 2020) framework with a CPU backend. We first conduct a comprehensive evaluation of state-of-the-art (SOTA) LLMs on MobileKernelBench. Our experimental results show that, due to limited framework-specific knowledge and insufficient optimization capability, LLM-generated kernels achieve performance parity with native framework implementations in at most 16.3% of benchmark cases, illustrated in Fig. 1 (a). Strikingly, more than

Table 1: **Comparison between server-side and mobile-side computing.** Unlike the server side, which utilizes unified CUDA-based backends on high-performance GPUs, the mobile side faces a fragmented landscape with diverse compute backends primarily optimized for inference tasks.

Dimension	Server Side	Mobile Side
Hardware	CPU, GPU, TPU, NPU (*ASIC*)	CPU, GPU, NPU (*SoC*)
Backend	CUDA, TensorRT, ROCm	CPU, OpenCL, Vulkan, CoreML, Metal
Language	CUDA, Triton, PTX	C++, Assembly
Feature	Homogeneous	Heterogeneous & Fragmented
Frameworks	PyTorch, TensorFlow, JAX	MNN, NCNN, TFLite, CANN
Workload	Training & Inference	Inference
Resource	Unlimited	Limited (Power/Mem/Band)

54.7% of the generated kernels fail at the compilation stage, primarily driven by hallucinated APIs or invalid framework usage, revealing a critical lack of grounding in framework-specific logic. Notably, only a small fraction of kernels achieve measurable speedups over the baseline. Subsequently, we apply standard training strategies, including LoRA (Hu et al., 2022) and GRPO (Shao et al., 2024), yet observe negligible improvements. We attribute these persistent failures to the scarcity of high-quality data within specific mobile inference frameworks. This data poverty creates a severe deficit in domain-specific knowledge, spanning framework specifications, optimization heuristics, and functional definitions, preventing LLMs from mastering this highly specialized domain.

To further explore the potential of LLMs in mobile kernel generation and address their identified limitations, we propose the Mobile Kernel Agent (**MoKA**), a specialized multi-agent system tailored for mobile kernel development. The MoKA follows a multi-round plan-and-execute paradigm, consisting of a code agent responsible for operator generation and two planning agents that respectively formulate execution strategies for compilation, functional correctness verification, and performance optimization. These agents are equipped with repository-aware and information parsing tools, enabling them to access and reason over realistic deployment signals. As illustrated in Fig. 1 (b), when evaluated on MobileKernelBench, the MoKA substantially outperforms the baseline and establishes SOTA performance among all tested LLMs, with 93.7% of kernels achieving successful compilation and 27.4% delivering measurable performance speedups. This initiative advances the frontier of mobile kernel generation, offering significant insights into enhancing LLM capabilities within such highly specialized domains.

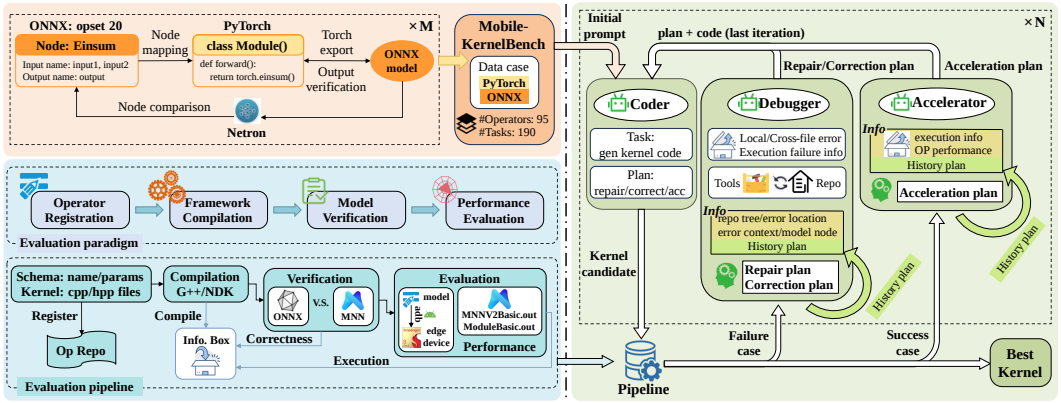
In summary, our contributions are as follows:

- We present the first systematic study extending automated kernel generation to the mobile domain, identifying three distinguishing features of mobile development and revealing the fundamental gap between server-side and mobile-side development.
- We introduce **MobileKernelBench**, a comprehensive evaluation system comprising a benchmark and an automated pipeline, facilitating systematic mobile kernel evaluation and granular information capture.
- We propose **MoKA**, a multi-agent system designed to autonomously navigate the complexities of API usage and performance optimization in data-scarce environments.
- We conduct extensive experiments on MobileKernelBench, revealing that while standard LLMs suffer from severe hallucinations and compilation failures, MoKA effectively overcomes these barriers, achieving SOTA performance over other LLMs.

2 Related Work

2.1 LLM for Kernel Generation

LLMs have shown proficiency in general code generation (Guo et al., 2024; Hui et al., 2024; OpenAI, 2025; Anthropic, 2025), and recent works have focused on leveraging them for high-performance computing (Waghjale et al., 2024; Huang et al., 2024). To systematically assess these capabilities, benchmarks such as KernelBench (Ouyang et al., 2025), MultiKernelBench (Wen et al., 2025), and TritonBench (Li et al., 2025a) have been developed, evaluating both correctness and efficiency across diverse platforms and paradigms. Recognizing the limitations of direct prompting,



(a) MobileKernelBench **(b) MoKA**

Figure 2: **Overview of our proposed framework.** The system consists of two core components: **(a) MobileKernelBench**, which establishes the evaluation environment by integrating a target-driven data curation process with an automated, hardware-in-the-loop evaluation pipeline; and **(b) MoKA**, a multi-role agentic system where Coder, Debugger, and Accelerator agents collaborate to iteratively generate and refine kernels based on feedback from the benchmark.

subsequent studies have integrated iterative feedback loops to refine kernel quality. One line of research, including Kevin (Baronio et al., 2025), AutoTriton (Li et al., 2025b), CUDA-L1 (Li et al., 2025c), and CUDA-L2 (Su et al., 2025), uses reinforcement learning (RL) or supervised fine-tuning (SFT) to optimize kernel generation. Alternatively, agentic frameworks such as Astra (Wei et al., 2025), EvoEngineer (Guo et al., 2025), and CudaForge (Zhang et al., 2025) employ collaborative role-playing or Coder-Judge architectures to ground reasoning in hardware specifications. However, these efforts predominantly focus on server-grade GPUs and CUDA, leaving the distinct constraints and fragmented ecosystems of mobile and edge computing largely unexplored.

2.2 Mobile Inference Engine

The Open Neural Network Exchange (ONNX) (Bai et al., 2019) serves as the de facto standard for model interoperability. By defining a unified intermediate representation and operator schema, ONNX decouples model architecture from specific training environments like PyTorch (Paszke et al., 2019) and TensorFlow (Abadi et al., 2016). This standardization is particularly critical for benchmarking, as it establishes a universal functional definition for operators, ensuring that evaluations focus on implementation capability rather than framework discrepancies. For on-device execution, bridging the gap between abstract representation and efficiency remains challenging. While compiler-based stacks like TVM (Chen et al., 2018) offer automation, library-based inference engines such as MNN (Jiang et al., 2020) and NCNN (ncnn contributors, 2017) are often preferred in industry for their lightweight integration. Achieving high performance in these libraries requires meticulous hardware-aware optimizations, including dynamic algorithmic selection and specialized memory layouts. Consequently, manually implementing operators involves managing complex low-level details like data packing and register allocation, creating a significant barrier that motivates the need for automated, expert-level code generation approaches.

3 MobileKernelBench

To bridge the gap between existing server-centric benchmarks and the specialized requirements of on-device inference, we introduce **MobileKernelBench**, the first benchmark dedicated to cross-framework kernel implementation for mobile deployment. As detailed in Tab. 2, our benchmark differs from previous work by focusing on the ONNX standard and incorporating a mobile kernel development evaluation pipeline, which verifies the full lifecycle of on-device integration.

3.1 Data Curation

Motivated by the mobile ecosystem's reliance on ONNX as the standard interchange format, our benchmark prioritizes comprehensive coverage of the ONNX operator set (Opset). To ensure compatibility with existing kernel generation benchmarks and standard LLM evaluation protocols, we encapsulate these operators within a PyTorch-based task format. To construct a usable, diverse, and robust dataset, we execute a systematic curation pipeline consisting of three stages.

Table 2: **Comparison between KernelBench and MobileKernelBench.** Compared to KernelBench, MobileKernelBench emphasizes operator diversity, covering the majority of common operator types in ONNX Opset 20. The (PyTorch, ONNX) pairs format facilitates kernel evaluation across different frameworks, targeting both framework adaptation and optimization on mobile devices.

Feature	KernelBench	MobileKernelBench (Ours)
Diversity	43 (Level 1)	95
Tasks	250	190
Format	PyTorch Models	(PyTorch, ONNX) Pairs
Domain	GPUs	Mobile SoCs (CPU, Adreno, etc.)
Target	Optimization ONLY	Framework Adaptation & Optimization.

Standard-aligned operator collection. We ground our benchmark in the ONNX Opset version 20 to ensure compatibility with modern inference engines. From the full ONNX operator catalog, we select operators according to two principles: diversity and generality. To ensure diversity, we include operators from a wide range of categories to reflect the heterogeneous computation patterns in neural network inference. To ensure generality, we prioritize operators that are widely used in practice, including those commonly appearing in canonical neural network architectures and those already supported by mainstream mobile inference frameworks such as MNN (Jiang et al., 2020) and NCNN (ncnn contributors, 2017). Following these criteria, we select 95 representative operators that capture the core workloads of on-device inference while keeping the benchmark compact.

Multi-dimensional task expansion. A single operator implementation is often insufficient to capture the complexity of varying attributes and input configurations. To rigorously evaluate the generalization capability of LLMs, we expand the 95 selected operators into 190 specific tasks through a task expansion strategy. Instead of relying on default parameters, we systematically mutate the PyTorch model definitions to cover diverse code paths and edge cases. Specifically:

- **Attribute variation:** We modify key attributes that directly affect control-flow behavior. For example, in reduction operators such as `ArgMax` and `ReduceSum`, we vary the `keep_dims` parameter and the reduction axis to evaluate the model's ability to handle output shape inference and loop reduction logic.
- **Dimensionality transformation:** We extend input tensors from standard 2D matrices to lower or higher-dimensional forms. For matrix multiplication operators including `MatMul` and `Gemm`, this requires the generated kernel to correctly handle broadcasting semantics and batch stride arithmetic, significantly increasing the implementation difficulty.

Target-driven construction strategy. To ensure the expanded tasks are valid and align with deployment standards, we employ a target-driven construction strategy. We actively construct paired (PyTorch, ONNX) instances as illustrated in Fig. 2(a) and filter them through two criteria:

- **Output fidelity:** The implemented PyTorch module must yield results identical to the target ONNX operator. We verify this by cross-referencing the execution of the PyTorch code in eager mode against the ONNXRuntime [1] execution of the exported model.
- **Topology alignment:** The PyTorch implementation must be converted to the precise target ONNX operator node rather than a composite subgraph. We validate this by inspecting the exported graph topology with Netron [2], iteratively refining the source code until the graph structure aligns with the canonical ONNX definition.

The resulting benchmark comprises 190 tasks covering 95 distinct operators, taxonomized into 12 categories in Tab. 3. This set spans diverse computational patterns, ranging from memory-bound element-wise operations to compute-bound contractions like convolutions.

3.2 Evaluation Pipeline

Evaluating mobile kernels presents a unique challenge: code generation occurs on host machines, while verification must be executed on resource-constrained edge devices. To address this, we propose a generic evaluation protocol for mobile kernels, which is instantiated through an automated, end-to-end pipeline as shown in Fig. 2(a). For more detailed configurations, see Sec. D

[1] https://github.com/microsoft/onnxruntime

[2] https://github.com/lutzroeder/netron

Table 3: **Overview of MobileKernelBench.** This benchmark comprises 190 tasks derived from 95 primitive operators. These operators are classified into 12 categories, encompassing common operators found in the ONNX ecosystem. A primitive operator may yield multiple distinct tasks based on differences in input shapes or parameter settings.

Categories	Representatives	#Operators	#Tasks
Unary	exp, sign, ceil, floor	11	11
Binary	add, div, mod	7	8
Trigonometry	sin, acos, atanh, tan	12	12
Activation	hardsigmoid, softmax, celu, relu	6	12
Normalization	batchnorm, layernorm, instancenorm	3	12
Pooling	maxpool, averagepool	4	15
Convolution	conv2d, convtranspose,	2	21
Matrix	einsum, gemm, matmul, det	4	19
Reduction	reducemin, reduceprod, reducesum	10	35
Tensor	reshape, concat, topk, tile, slice	19	28
Logic	and, bitwisexor, equal	11	13
Others	RNN, LSTM, STFT, RoiAlign	4	4
Total		**95**	**190**

Evaluation paradigm. We design a standardized four-stage protocol comprising *Operator Registration*, *Framework Compilation*, *Model Verification*, and *Performance Evaluation*. This paradigm explicitly decouples operator implementation from runtime execution. By formalizing this separation, we ensure the evaluation remains reproducible and accurately simulates the deployment workflow of real-world mobile applications.

Pipeline instantiation on MNN. We instantiate this paradigm within the MNN framework, automating the operator lifecycle through four specific stages: (1) **Automated registration.** Upon code generation, the pipeline parses the output and executes an injection strategy. It locates the target source path and hot-swaps the existing implementation with the generated code, ensuring seamless registration into the global operator factory without manual intervention. (2) **Framework compilation.** The pipeline invokes the CMake build system to compile the modified MNN library. This stage acts as a strict filter for syntactic correctness and API validity, where errors trigger immediate failure termination. (3) **Functional verification.** Successfully compiled operators undergo differential testing. We establish a ground truth baseline using the reference ONNX model with randomized inputs. The pipeline converts the ONNX model to MNN format and compares the MNN inference output against the baseline, enforcing a strict numerical tolerance for validation. (4) **On-device performance benchmarking.** We employ a cross-compilation and remote execution workflow for real-world efficiency evaluation. The modified source is cross-compiled via the Android NDK, and the resulting ARM64 binaries are deployed to the target device via a bridge interface. To ensure statistically robust measurements, the benchmarking executable follows a strict warm-up and multi-iteration protocol to mitigate system noise.

4 MoKA

We propose the Mobile Kernel Agent (MoKA), a specialized multi-agent framework designed to automate the lifecycle of operator implementation, debugging, and optimization for on-device inference engines. As shown in Fig. 2 (b), MoKA follows an iterative *plan-and-execute* paradigm. In each iteration, the agent generates an operator candidate that probes the deployment environment, while the evaluation pipeline returns structured feedback including compilation status, correctness, and performance metrics. This feedback loop drives the agents to refine their planning strategies, enabling progressive convergence toward high-quality, deployment-ready operators.

4.1 Agent Collaboration Design

The MoKA architecture decomposes the implementation task into three specialized roles that collaborate via a shared history memory.

Coder. The Coder serves as the sole actuator of the system, responsible for synthesizing C++ source code. In the initial iteration, it generates a draft implementation based on the task description. In subsequent steps, it acts as an executor that translates high-level strategies provided by planning agents into concrete code modifications, ensuring adherence to the specified repair or optimization logic. **Debugger.** Activated upon pipeline failures, the Debugger handles two distinct error modes. For build failures, it initiates a *Compilation Diagnosis (Repair Plan)* by utilizing repository-aware tools to interpret compiler diagnostics and retrieve cross-file context, thereby formulating plans to fix syntactic or dependency errors. For incorrect outputs, it initiates a *Functional Correction (Correction Plan)* by employing model parsing tools to align the implementation with the ONNX definition, identifying semantic discrepancies to generate functionality-preserving fixes. **Accelerator.** Once an operator passes functional verification, the Accelerator pushes performance boundaries. It leverages a performance parser to extract fine-grained execution metrics such as backend selection and threading efficiency. Combined with historical data, it proposes an *Acceleration Plan* to optimize memory access patterns or computational logic without compromising correctness.

Reflective memory. Crucially, both the Debugger and Accelerator utilize a shared history mechanism. By maintaining a trace of past strategies and their outcomes, the agents perform self-reflection to avoid repetitive mistakes and prune ineffective optimization paths.

4.2 Agentic Toolset

To ground agent reasoning within the rigid constraints of the deployment environment, we equip them with two categories of domain-specific tools.

Repository-aware tools. To address the lack of structural awareness in generic LLMs, we implement a *Repository Tree Builder* that efficiently constructs a hierarchical view of the codebase, facilitating the resolution of dependency errors. Additionally, an *Error Extractor* powered by *tree-sitter-cpp* (Associates et al., 2025) parses compiler logs to pinpoint exact error locations and extract surrounding context. These tools enable the Debugger to effectively distinguish between local syntax errors and complex cross-file inconsistencies. **Information parsing tools.** To bridge static graphs and dynamic execution, a *Model Parser* serializes both reference ONNX nodes and converted MNN operators into a unified structured representation, providing evidence for semantic mismatches. Furthermore, a *Performance Parser* processes raw profiling logs (e.g., backend usage, execution time) into structured indicators. This assists the Accelerator in identifying bottlenecks, such as suboptimal data layouts or thread contention, to formulate backend-consistent optimization strategies.

5 Experiments

In this section, we conduct a comprehensive evaluation on MobileKernelBench to assess the performance of LLMs in mobile kernel generation. In Sec. 5.2, we first evaluate leading open-source and closed-source models accessed directly via cloud service APIs. Furthermore, to investigate the efficacy of standard fine-tuning strategies for this task, we implement classic LoRA fine-tuning and GRPO training. Finally, we evaluate the performance of our proposed MoKA in Sec. 5.3.

5.1 Experiment Setups

Evaluation environment. We utilize MNN (v3.2.2 [3]) as the foundational inference engine, specifically targeting its CPU backend for operator registration and execution (see Sec. B for registration details). All evaluations are conducted on a Xiaomi 13 smartphone powered by the Qualcomm Snapdragon 8 Gen 2 platform (Qualcomm Technologies, Inc., 2022), a representative high-end mobile SoC equipped with an 8-core Kryo CPU and LPDDR5X memory.

Evaluation metrics. We assess performance on MobileKernelBench using three metrics: (1) **Compilation success rate (CSR)**, the percentage of tasks where generated operators successfully pass the build process. (2) **Functional correctness rate (FCR)**, the proportion of tasks that pass functional verification against the ONNX baseline. (3) $\mathbf{fast}_p$ (Ouyang et al., 2025), this metric quantifies the percentage of tasks achieving a speedup greater than a threshold p relative to the native MNN implementation. For methods that involve iterative refinement or generate multiple candidates, we report metrics based on the best-performing operator for each task.

[3] https://github.com/alibaba/MNN/releases/tag/3.2.2

Table 4: **Performance comparison of SOTA LLMs on MobileKernelBench.** The best and second-best results are highlighted in **bold** and underlined respectively. Some model names are abbreviated for layout purposes, with full identifiers provided in Sec. 5.2.

Baseline	CSR	FCR	fast_p		
	(%)	(%)	**0.5**	**1.0**	**1.5**
Claude-Sonnet-4.5	46.3	**34.2**	**31.1**	**16.3**	**4.7**
GPT-5	**47.4**	33.2	26.3	13.7	4.2
Gemini-2.5-Flash	20.5	15.8	13.2	6.8	0.5
Llama-3.1-405B	11.6	6.3	5.3	4.2	1.1
DeepSeek-R1	15.3	10.0	8.4	3.2	0.5
Qwen3-235B	18.9	13.7	11.6	6.3	2.1

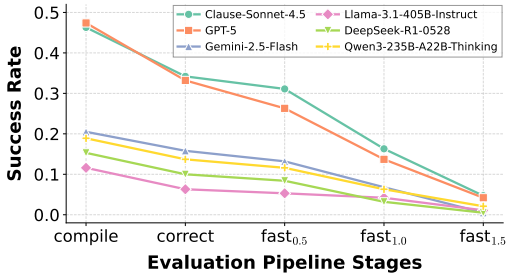


Figure 3: **Success rate degradation across evaluation stages.** The plot illustrates the performance drop of each model as the evaluation criteria become stricter, from compilation to functional correctness and varying levels of performance optimization.

LoRA training setups. We fine-tune Qwen-32B via LLaMA-Factory (Zheng et al., 2024) using LoRA (rank 64) and ZeRO stage-3 (Rajbhandari et al., 2020) on 8 A100 GPUs. The model is trained for 2 epochs with a batch size of 1 and gradient accumulation steps of 2. The training dataset follows the Alpaca format, pairing GPT-5 generated descriptions with MNN implementations of 74 ONNX operators. A subset of 20 operators and 36 *MobileKernelBench* tasks is held out for testing.

GRPO training setups. We explore RL using GRPO (Shao et al., 2024) implemented via the verl (Sheng et al., 2024) framework, with Qwen3-4B-Instruct-2507 as the policy model. Our *MobileKernelBench* is partitioned into a training set of 150 samples and a test set of 40 samples using stratified sampling to ensure a balanced representation of operator categories. Based on the reward scheme from Kevin (Baronio et al., 2025), we add an intermediate compilation reward to avoid extremely sparse rewards in early training stages. Training runs on two A100 GPUs for 40 steps, using a global batch size of 30, a learning rate of 1e-6, a group size of 5, and a context length of 8192 tokens.

5.2 Baseline Evaluation

We benchmark six prevalent LLMs, spanning proprietary leaders (OpenAI GPT-5 (OpenAI, 2025), Anthropic Claude-Sonnet-4.5 (Anthropic, 2025), Google Gemini-2.5-Flash (Google DeepMind, 2025)) and open-source frontiers (LLaMA-3.1-405B-Instruct (Dubey et al., 2024), DeepSeek-R1-0528 (Guo et al., 2024), Qwen3-235B-A22B-Thinking-2507 (Yang et al., 2025)). All models are evaluated using a standardized initial prompt (see Sec. C.1) under default settings via API endpoints.

The quantitative results presented in Tab. 4 and Fig. 3 demonstrate a substantial gap between code generation capabilities and deployment readiness across all evaluated models. Although leading proprietary models, such as Claude-Sonnet-4.5 and GPT-5, achieve compilation success rates of approximately 47%, their performance declines significantly when strict functional verification criteria are applied. Furthermore, open-source models demonstrate considerably lower proficiency, with functional correctness rates plateauing between 6.3% and 13.7%, which underscores the complexity of synthesizing valid operators for low-resource frameworks. The fast_p metrics highlight the difficulty in achieving high-performance optimization. Even for the top-performing Claude-Sonnet-4.5, only 16.3% of the generated operators match or exceed the baseline speed, with just 4.7% realizing a significant speedup ($> 1.5\times$). These findings substantiate the hypothesis that base large language models lack the intrinsic ability to discover hardware-efficient implementation strategies in the absence of external guidance or feedback.

We further dissect model performance across different operator categories, as illustrated in Fig. 4. For computationally lightweight operations such as activation functions, most models exhibit strong performance, with Claude-Sonnet-4.5 achieving a functional correctness rate exceeding 70%. In contrast, complex operators present a prohibitive challenge. Gemini-2.5-Flash and the leading open-source models fail to produce any functionally correct kernels for convolution tasks. Unexpectedly, GPT-5 and DeepSeek-R1-0528 display specific competence in matrix operations, achieving functional correctness rates of 52.6% and 31.6%, respectively. This observation suggests that these models may have retained effective algorithmic structures for general matrix multiplication during their pre-training phases, despite having lower overall performance averages.

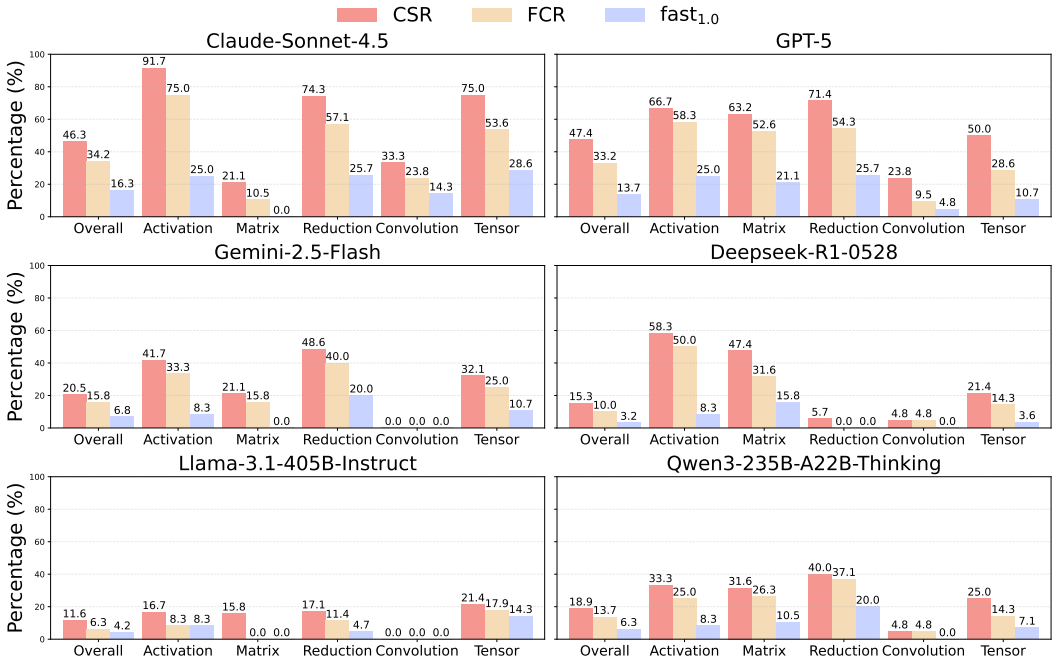


Figure 4: **Fine-grained performance of LLMs across different operator categories.** We visualize the evaluation metrics for five representative operator types. The results highlight significant disparities in model capabilities when handling operators with varying levels of algorithmic complexity.

Supervised fine-tuning result by LoRA. As shown in Tab. 5, the base Qwen3-32B model demonstrates limited proficiency in this domain-specific task. Although SFT resulted in marginal improvements, specifically increasing the compilation success rate to 25.0% and correctness to 18.5%, it failed to yield any gains in the strict $\text{fast}_{1.5}$ efficiency metric. We attribute this limitation primarily to the inherent data scarcity characterizing mobile operator development. Given the constrained volume of available training samples, supervised learning proved insufficient for encoding the deep, framework-specific semantic knowledge required for MNN implementation. In contrast, our MoKA circumvents this bottleneck by actively retrieving optimization strategies via external tools. This distinction underscores that for low-resource, expert-domain tasks, an agentic framework capable of dynamic context retrieval and iterative reasoning constitutes a significantly more robust solution than static model fine-tuning.

Reinforcement learning result by GRPO. As shown in Tab. 5, the application of GRPO to the Qwen3-4B-Instruct-2507 model results in a significant 15.0% improvement in the compilation success rate, effectively validating that our hierarchical reward design can successfully guide the model to comply with strict syntactic constraints. However, this method encounters limitations in terms of functional logic and performance optimization. The functional correctness rate only increases marginally by 2.5%, and the high-performance metric $\text{fast}_{1.0}$ remains unchanged at 5.0%. This stagnation indicates that although RL can align the model's output format, it fails to develop the complex reasoning skills needed to identify hardware-efficient strategies in smaller models. The superior performance of our MoKA demonstrates that iterative refinement is significantly more effective than training-time policy optimization for mobile operator generation.

5.3 MoKA Results

We initialize the MoKA using Claude-Sonnet-4.5 and perform N=10 iterations. To isolate the benefits of our agentic workflow from simple sampling diversity, we compare it against a pass@10 baseline where the same model is queried ten times. As shown in Tab. 5, while the pass@10 strategy yields a moderate improvement over baseline, the MoKA demonstrates superior performance across all metrics. Specifically, it achieves a functional correctness rate of 75.3%, surpassing the single-query and pass@10 baselines by 41.1% and 27.4%, respectively. Most notably, in the speedup evaluation, the MoKA attains a 27.4% success rate at the challenging $\text{fast}_{1.5}$ threshold, whereas

Table 5: **Comprehensive performance evaluation on MobileKernelBench.** We compare our proposed MoKA against standard prompting, SFT, and RL methods. CSR and FCR represent compilation success rate and functional correctness rate, respectively. Bold values indicate the best performance, and values in parentheses denote the absolute gains over the corresponding baselines.

Method	CSR (%)	FCR (%)	fast_p 0.5	fast_p 1.0	fast_p 1.5
Supervised Fine-Tuning (SFT)					
Qwen3-32B	16.7	13.9	8.3	2.8	0.0
Qwen3-32B-LoRA	25 (+8.3)	18.5 (+5.6)	11.1 (+2.8)	5.6 (+2.8)	0.0 (–)
Reinforcement Learning (RL)					
Qwen3-4B-Instruct-2507	10.0	5.0	5.0	5.0	0.0
Qwen3-4B-Instruct-2507-GRPO	25.0 (+15.0)	7.5 (+2.5)	7.5 (+2.5)	5.0 (–)	0.0 (–)
Agentic Workflow					
Claude-Sonnet-4.5 (Base)	46.3	34.2	31.1	16.3	4.7
Claude-Sonnet-4.5 (@10)	62.1	47.9	41.6	20.5	5.3
MoKA (ours)	**93.7** (+47.4)	**75.3** (+41.1)	**62.6** (+31.5)	**46.8** (+30.5)	**27.4** (+22.7)

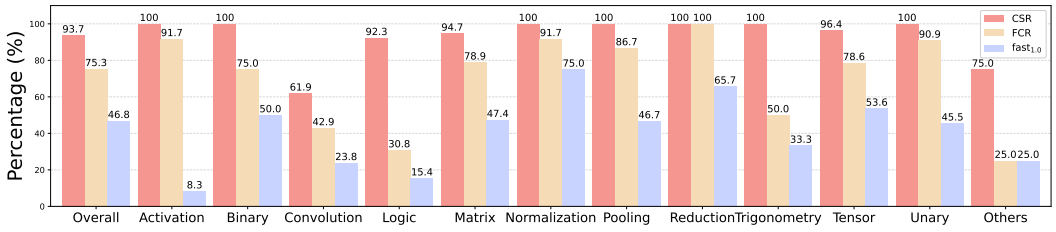


Figure 5: **Fine-grained performance of MoKA across different operator categories.** The agent demonstrates robust generalization across diverse operator types, achieving high correctness on structurally simpler operations (e.g., activation, normalization) while maintaining competitive performance on complex tasks like convolution and matrix operations.

baselines fail to exceed 6%. This confirms that our iterative feedback loop enables the generation of not only correct but also highly efficient code.

Fig. 5 further breaks down performance by operator category. The MoKA achieves a 100% compilation success rate in seven categories and yields substantial correctness gains on complex tasks such as matrix ($> 7\times$) and convolution (nearly $2\times$). These results indicate that the Debugger effectively leverages repository context to resolve hallucinations and dependency errors. Moreover, the average speedup of nearly $3\times$ across successful kernels validates the efficacy of the Accelerator in identifying hardware-specific optimizations.

6 Conclusion

In this work, we investigate the automation of mobile kernel development using Large Language Models (LLMs). We introduce MobileKernelBench, a robust system coupled with an automated pipeline to holistically evaluate compilation success, functional correctness, and on-device performance. Our comprehensive evaluation of both off-the-shelf and fine-tuned models reveals that LLMs struggle with domain-specific mobile implementation, a failure we attribute to the knowledge deficits caused by ecosystem fragmentation and data scarcity. To bridge this gap, we propose MoKA, a framework that decomposes kernel refinement into cooperative agents for generation, debugging, and acceleration. MoKA achieves SOTA results, validating the efficacy of agentic optimization in this specialized domain. Our findings confirm that with principled methodological design, LLMs can effectively assist human developers in mobile kernel engineering. Future research directions include extending these capabilities to low-level optimizations such as multi-threaded NEON and inline assembly, exploring diverse hardware backends and frameworks beyond MNN, and potentially automating the generation of entire deployment frameworks.

REFERENCES

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, 2016.

Anthropic. Introducing Claude Sonnet 4.5. https://www.anthropic.com/news/claude-sonnet-4-5, September 2025. Accessed: 2026-01-09.

Applied Research Associates, GitHub, astral sh, Disjunctive, sensmetry, AWS, University of Graz, Abacus AI, GuildEducationInc, Chime Systems Inc., Intel-Corporation, River Point Technology, and slashwhy. tree-sitter/tree-sitter: An incremental parsing system for programming tools, 2025. URL https://github.com/tree-sitter/tree-sitter.

Junjie Bai, Fang Lu, Zhang Ke, et al. ONNX: Open neural network exchange. https://github.com/onnx/onnx, 2019. Accessed: 2026-01-09.

Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. Kevin: Multi-turn rl for generating cuda kernels. *arXiv preprint arXiv:2507.11948*, 2025.

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 *OSDI*, 2018.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv e-prints*, 2024.

Google DeepMind. Gemini Models – Next Generation AI Systems. https://www.deepmind.google/models/gemini/, 2025. Accessed: 2026-01-09.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

Ping Guo, Chenyu Zhu, Siyuan Chen, Fei Liu, Xi Lin, Zhichao Lu, and Qingfu Zhang. Evoengineer: Mastering automated cuda kernel code evolution with large language models. *arXiv preprint arXiv:2510.03760*, 2025.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. In *ICLR*, 2022.

Dong Huang, Yuhao Qing, Weiyi Shang, Heming Cui, and Jie M Zhang. Effibench: Benchmarking the efficiency of automatically generated code. In *NeurIPS*, 2024.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Xiaotang Jiang, Huan Wang, Yiliu Chen, Ziqi Wu, Lichuan Wang, Bin Zou, Yafeng Yang, Zongyang Cui, Yu Cai, Tianhang Yu, et al. Mnn: A universal and efficient inference engine. In *MLSys*, 2020.

Jianling Li, Shangzhan Li, Zhenye Gao, Qi Shi, Yuxuan Li, Zefan Wang, Jiacheng Huang, Wang-Haojie WangHaojie, Jianrong Wang, Xu Han, et al. Tritonbench: Benchmarking large language model capabilities for generating triton operators. In *ACL*, 2025a.

Shangzhan Li, Zefan Wang, Ye He, Yuxuan Li, Qi Shi, Jianling Li, Yonggang Hu, Wanxiang Che, Xu Han, Zhiyuan Liu, et al. Autotriton: Automatic triton programming with reinforcement learning in llms. *arXiv preprint arXiv:2507.05687*, 2025b.

Xiaoya Li, Xiaofei Sun, Albert Wang, Jiwei Li, and Chris Shum. Cuda-l1: Improving cuda optimization via contrastive reinforcement learning. *arXiv preprint arXiv:2507.14111*, 2025c.

The ncnn contributors. ncnn, June 2017. URL https://github.com/Tencent/ncnn.

OpenAI. Introducing gpt-5. https://openai.com/index/introducing-gpt-5/, Aug 2025. Accessed: 2026-01-09.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Re, and Azalia Mirhoseini. Kernelbench: Can LLMs write efficient GPU kernels? In *ICML*, 2025.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, 2019.

Qualcomm Technologies, Inc. Snapdragon 8 Gen 2 Mobile Platform: Product Brief. PDF, 2022. URL https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/Snapdragon-8-Gen-2-Product-Brief.pdf. Product brief (2 pages).

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC*, 2020.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*, 2024.

Songqiao Su, Xiaofei Sun, Xiaoya Li, Albert Wang, Jiwei Li, and Chris Shum. Cuda-l2: Surpassing cublas performance for matrix multiplication through reinforcement learning. *arXiv preprint arXiv:2512.02551*, 2025.

Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, et al. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025.

Siddhant Waghjale, Vishruth Veerendranath, Zhiruo Wang, and Daniel Fried. Ecco: Can we improve model-generated code efficiency without sacrificing functional correctness? In *EMNLP*, 2024.

Anjiang Wei, Tianran Sun, Yogesh Seenichamy, Hang Song, Anne Ouyang, Azalia Mirhoseini, Ke Wang, and Alex Aiken. Astra: A multi-agent system for GPU kernel performance optimization. In *NeurIPS 2025 Fourth Workshop on Deep Learning for Code*, 2025.

Zhongzhen Wen, Yinghui Zhang, Zhong Li, Zhongxin Liu, Linna Xie, and Tian Zhang. Multikernelbench: A multi-platform benchmark for kernel generation. *arXiv eprints, pp. arXiv–2507*, 2025.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, et al. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models. *arXiv preprint arXiv:2508.06471*, 2025.

Zijian Zhang, Rong Wang, Shiyang Li, Yuebo Luo, Mingyi Hong, and Caiwen Ding. Cudaforge: An agent framework with hardware feedback for cuda kernel optimization. *arXiv preprint arXiv:2511.01884*, 2025.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, 2024.

A OVERVIEW

This appendix provides implementation details and experimental configurations to facilitate the reproducibility of MoKA. We begin by defining the taxonomy of MNN operators in Sec. B, which structures our code generation strategy based on operator implementation mechanisms. Building on this, Sec. C presents the context-aware prompt templates designed for the Coder, Debugger, and Accelerator agents. To enable rigorous validation, Sec. D outlines the construction of our evaluation pipeline, covering repository restructuring, incremental cross-compilation, and on-device benchmarking protocols. We then elaborate on the GRPO training methodology in Sec. E, detailing the compound reward formulation and the parallelized remote-mobile infrastructure used to handle sparse rewards and hardware constraints. Finally, Sec. F provides a granular case study of the LayerNorm2D operator, illustrating the iterative optimization trajectory and the diverse hardware-aware strategies deployed by the agent to achieve significant speedups.

B MNN OPERATOR INFORMATION

To ensure seamless integration with the MNN framework, we categorize all the operators into three distinct types based on their implementation mechanisms within MNN's architecture. This taxonomy aligns with the framework's operator lifecycle and dictates the specific source files targeted for code generation and replacement.

- **Atomic operators:** This category encompasses fundamental operators that require direct, hardware-aware implementation on the CPU backend. These operators typically involve intensive numerical computation and cannot be trivially decomposed.
- **Geometric operators:** This category includes operators that can be mathematically expressed as coordinate transformations of input tensors.
- **Composite operators:** This category refers to high-level operators that do not have a direct one-to-one mapping in the MNN backend but can be represented as a composition of existing atomic operators.

Atomic operators require the synthesis of paired declaration (`.hpp`) and implementation (`.cpp`) files to define execution classes, while geometric and composite operators are encapsulated within a single source (`.cpp`) file. This structured classification enables our evaluation pipeline to automatically locate the corresponding C++ source files for injection and compilation, ensuring a robust and automated benchmarking process.

C EXPERIMENT PROMPTING DETAILS

C.1 INITIAL PROMPT FOR CODER

The Coder begins by querying the LLMs to generate C++ operator implementations. To bridge the gap between abstract PyTorch operator definitions and the architectural requirements of the MNN framework, we develop a dynamic, context-aware prompt generation mechanism. Instead of using a generic instruction, we construct a structured prompt tailored to the specific attributes of the target operator. Besides the PyTorch model definition of the target operator, the prompt also comprises the following key components:

- **System role and task definition:** We define the LLM's role as a mobile model deployment expert, with the explicit goal of converting a PyTorch model into a C++ operator for execution on the MNN CPU backend.
- **Constraint specification:** We enforce strict architectural constraints on the generated code, including adherence to internal APIs, implementation of required lifecycle methods (e.g., onResize, onExecute), and prescribed file naming and organization conventions.
- **Target operator context:** A key innovation of our pipeline is the dynamic injection of framework-specific knowledge. Based on the category of the target operator (e.g., unary, binary, reduction, or convolution), the relevant C++ header files are automatically retrieved and embedded into the prompt.

- **One-shot example:** To guide the code structure, we provide a paired example of PyTorch models and MNN C++ implementation, which aligns with the target implementation mechanism in the MNN library, corresponding to atomic, geometric, or composite operators.

To provide a concrete illustration of this context-aware generation mechanism, we present the full initial prompt used for the `ArgMax` operator below.

```
You are an expert in model deployment, proficient in PyTorch and C++ programming, and familiar
      with the coding style of the MNN framework. You will be given a PyTorch model which will
      be exported as an ONNX graph and then converted into an MNN computation graph. Your task
      is to write C++ code that implements and accelerates the operators from this model for
     MNN's CPU backend. Note that:
- Understand the example thoroughly and think carefully before you write the code.
- Provide only the code file as your final answer, without any explanations or comments.
- Your code must adhere to the supported API surfaces, invoking only official functions and
     members when using MNN C++ interfaces( e.g. Math, Tensor, VARP, Matrix) and flatbuffers
     library.
- Each file name must appear as the very first line inside its corresponding code block. If
     provide cpu backend implement, provide the hpp code and cpp code seperately.
- Implement the operator's computation logic in a self-contained manner. Minimize coupling to
     MNN internals and 3rd party libraries; call APIs only when strictly required.
- Implement methods include: the CPU backend implement which handles numerical computation for
      operators by memory management and instruction-level optimization, the geometry
     computation which manages data layout and memory mapping and is used for operators that
     change tensor shapes or memory arrangements, the combinator implementation which builds
     new operator functions by composing existing MNN operators.
- When write CPU backend operator, implement onResize and onExecute. In onResize, allocate the
      cache buffer using backend()->onAcquireBuffer(&mCache, Backend::DYNAMIC) and release it
     with backend()->onReleaseBuffer(&mCache, Backend::DYNAMIC), allowing the freed memory to
     be reused. In onExecute, perform necessary input validation to catch issues early. Return
      NO_ERROR upon successful execution.
- When write a Geometry backend operator, implement onCompute to construct the tensor regions
     and command sequence that describe how outputs are assembled from inputs, allocating any
     intermediate tensors as virtual slices so the runtime can later schedule the actual
     computation efficiently.
- When write a Combiner operator, implement the onExecute method which parses the ONNX node's
     inputs and attributes to construct an equivalent computational subgraph using MNN's
     Express API and returns the converted expression to complete the operator translation.

Here is the example:
===== Example Start =====
Given PyTorch model Det:
[PyTorch code for Det model...]

By using CPU backend implemetation, you should respond with the final answer with two files
     seperately:

CPUDet.hpp:
'''
#ifndef CPUDet_hpp
#define CPUDet_hpp
// ... [Includes and Class Definition]
class CPUDet : public Execution {
// ... [declarations for onResize, onExecute]
};
#endif
'''

CPUDet.cpp:
'''
#include "CPUDet.hpp"
// ... [Other Includes]

namespace MNN {
ErrorCode CPUDet::onResize(...) {
// ... [Buffer allocation logic]
return NO_ERROR;
}

ErrorCode CPUDet::onExecute(...) {
// ... [Complex determinant computation logic omitted]
return NO_ERROR;
```

```
}

class CPUDetCreator : public CPUBackend::Creator {
// ... [Creator implementation]
};
REGISTER_CPU_OP_CREATOR(CPUDetCreator, OpType_Det);
}
```
```
===== Example END =====

Now you are given the following PyTorch model:
```
```
import torch
import torch.nn as nn

class Model(nn.Module):
    """
    Simple model that performs Argmax over a specified dimension.
    """
    def __init__(self, dim: int):
        """
        Initializes the model with the dimension to perform argmax.

        Args:
            dim (int): The dimension to perform argmax over.
        """
        super(Model, self).__init__()
        self.dim = dim

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """
        Applies argmax over the specified dimension to the input tensor.

        Args:
            x (torch.Tensor): Input tensor.

        Returns:
            torch.Tensor: Output tensor with argmax applied, with the specified dimension
                removed.
        """
        return torch.argmax(x, dim=self.dim)

batch_size = 32
dim1 = 128
dim2 = 256

def get_inputs():
    x = torch.rand(batch_size, dim1, dim2)
    return [x]

def get_init_inputs():
    return [1]
```
```
Implement the CPU backend for the corresponding operator in this PyTorch model. You need to
     write the CPUArgMax.hpp and CPUArgMax.cpp files separately.
```

C.2 Prompt for Debugger

The Debugger is activated when the generated kernel code fails either during the compilation process or the functional correctness verification. We have designed two specialized prompt templates to address these distinct failure modes, ensuring the agent receives the relevant context needed to diagnose and resolve issues effectively.

When the MNN build system reports compilation errors, the constructed prompt includes:

- **Operator context:** A description of the target operator, including its name, functionality, and the PyTorch reference model.
- **Current implementation:** The full source code generated by Coder that caused the failure.

- **Structured error log:** A parsed dictionary of compilation errors, categorized into *in-place errors* (syntax or logic errors within the generated file) and *cross-file errors* (mismatches with external MNN APIs or definitions). For each error, we provide the file path, line number, specific error message, and the relevant code context.

The agent is instructed to analyze these errors and provide semantic suggestions for code correction. Crucially, the instructions constrain the Debugger to suggest modifications *only* for the current code fragment, treating external MNN files as immutable references. The output is required in a structured JSON format containing lists of suggestions for both in-place and cross-file errors. Here is the prompt template and the provided error information:

```
'''
Here is a brief description of the operator to be implemented in MNN:
operator information:{op_info}

To implement this operator in MNN, the following script code has been implemented:
{code_book}

Compiling error accurred during MNN operator compilation, the information is listed as follows
     :
{compile_error}

Please analysize the compiling error and give suggestions to improve the MNN operator code.
Note:
 - Only provide semantic suggestions (in text) to improve the code.
 - Make sure your suggestion cover all the errors listed above.
 - Only provide semantic suggestions (in text) to improve the code.
 - All suggestions must be simple and direct.
 - Any suggestion involving code changes must modify only the current code snippet itself.
 - Code from other files is correct and for reference only. Do not suggest any modifications
     to other code.
 - For inplace errors, focus on correcting the code snippets provided in the error contexts.
 - For cross-file errors, refer to the relevant code snippets(function call, parameter
     settings, function defination etc.) in the error contexts and adjust the implementation
     currently provided code accordingly.
 - TODO: headers error.

 Finally provide suggestions as follows:
 ```error_suggestion
{
    {
        "local_error_suggestion":[],
        "crossfile_error_suggestion":[],
    }
}
 ```
'''
```

```
compile_error:{
"opname":"opname",
"local_error":{
    "erorr1":{
        "erorr_file":"path/to/error_file1",
        "error_line":123,
        "error_message":"Description of error 123",
        "error_line":456,
        "error_message":"Description of error 456",
        "error_context":"Context or code snippet related to error"
    }},
"crossfile_error":{
    "erorr1":{
        "erorr_file":"path/to/error_file1",
        "error_line":123,
        "error_message":"Description of error 123",
        "error_line":456,
        "error_message":"Description of error 456",
        "error_context":"Context or code snippet related to error 1" #use tree-sitter to
             extract code snippet(the whole fuction or class that include the error line)
    },
```

```
    "erorr2":""
},
"other_error":"other_error"
}
```

If the kernel compiles successfully but fails the correctness verification against the ONNX baseline, a different prompt is generated containing:

- **Operator context & implementation:** The same operator description and current code implementation as in the compilation error scenario.
- **Execution error log:** Detailed runtime error messages or output mismatch information captured during the test execution.
- **Model topology comparison:** JSON representations of both the reference ONNX model and the converted MNN model. This allows the agent to inspect graph-level discrepancies, such as incorrect attribute mapping or tensor shape mismatches.

The prompt directs the Debugger to compare the MNN and ONNX model structures alongside execution errors to identify logical flaws. Similar to the compilation error scenario, the agent must provide straightforward and actionable semantic suggestions for fixing the implementation script, formatted as a JSON list of items. Here is the template:

```
'''
Here is a brief description of the operator to be implemented in MNN:
operator information:{op_info}

To implement this operator in MNN, the following script code has been implemented:
{code_book}

The code has passed the compile process, but there are functionality correctness issues during
      testing. The execute information is listed as follows:
{execute_error}

The json files of onnx model and mnn model are:
onnx information:{onnx_json} \n
mnn information: {mnn_json} \n

Analysize the execution error and compare the information provided, then give suggestions to
     correct the MNN implemetation script code.
Note:
 - Only provide semantic suggestions (in text) to improve the code.
 - All suggestions must be simple and direct.
 - Any suggestion involving code changes must modify only the current code snippet itself.
 - Code from other files is correct and for reference only. Do not suggest any modifications
      to other code.
 - Refer to the differences between the MNN and ONNX json files, and propose reasonable code
      modification suggestions to acheive functionality correctness.

 Finally provide suggestions as follows:
 ```functionality_suggestion
{
    {
        "suggestion1":"",
        "suggestion2":"",
        # More ...
    }
}
 ```
'''
```

C.3 Prompt for Accelerator

Once a generated kernel successfully passes both compilation and functional correctness verification, the Accelerator is engaged to further optimize its runtime performance. The prompt for this agent is designed to elicit a focused, high-impact optimization strategy rather than a broad list of potential improvements. The input context provided to the agent includes:

- **Operator context & implementation:** The same as Debugger.
- **Performance metrics:** The execution latency and relevant profiling data obtained from the on-device benchmarking of the current kernel.
- **Optimization history:** A record of previously attempted optimization strategies for this or similar operators. This is critical for preventing the agent from suggesting redundant or previously failed optimizations, ensuring efficient exploration of the solution space.

The instructions explicitly require the agent to identify exactly one primary performance bottleneck and propose exactly one corresponding optimization method expected to yield the largest speedup. Furthermore, the agent is directed to provide a concrete modification plan while keeping descriptions brief and technical. The output must be formatted as a JSON object containing three fields: bottleneck (diagnosis of the performance issue), optimization method (the proposed strategy), and modification plan (actionable steps for implementation). This structured output facilitates automated parsing and subsequent code generation cycles. Here is the prompt template:

```
'''
You are an expert in model deployment, proficient in PyTorch and C++ programming, and familiar
      with the coding style of the MNN framework. Your task is to analyse the performance
     bottlenecks of the following MNN operator code and propose optimisation methods to
     accelerate it.
Then identify **exactly one** highest-impact speed bottleneck, propose **exactly one**
     optimisation method and propose a modification plan.
Here is a brief description of the operator to be implemented in MNN:
operator information:{op_info}

To implement this operator in MNN, the following script code has been implemented:
{code_book}

Here is the current performance of the operator:
{performance}

Here is the history optimisation information of similar operators:
{history_optmz_info}

Requirements:
- Return **one and only one** optimisation method -- the largest expected speedup.
- Keep fields brief; avoid lists of alternatives, disclaimers, or generic advice.
- Avoid the totally same optimizations that have already been attempted in the history
     optimisation information.

Output format (JSON):
```json
{{
  "bottleneck": "<max 100 words>",
  "optimisation method": "<max 100 words>",
  "modification plan": "<max 100 words>"
}}
'''
```

C.4 Iterative Refinement Prompt for Coder

During the iterative optimization process, the Coder is re-engaged to modify the existing kernel implementation based on feedback from downstream agents. Depending on the state of the current kernel, the prompt is dynamically adjusted to focus either on code repair or performance optimization. The specific instructions diverge as follows:

- **Repair mode:** When the kernel fails compilation or correctness tests, the prompt incorporates the Repair Suggestions generated by the Debugger agent. The Coder is explicitly tasked with refining the code to resolve these specific errors.
- **Acceleration mode:** When the kernel is functionally correct but requires speedup, the prompt incorporates the Optimization Plan generated by the Accelerator Agent. The Coder is tasked with implementing the proposed algorithmic or memory-level optimizations to improve execution latency.
```

To ensure the output can be seamlessly integrated into the evaluation pipeline, strict formatting constraints are applied in both modes. The agent is required to generate only the C++ code without any accompanying natural language explanations or markdown commentary, which facilitates direct file overwriting and subsequent compilation cycles.

D EVALUATION PIPELINE BUILDING FROM MNN

To instantiate our evaluation pipeline, we select MNN as the target framework for on-device inference. We begin by restructuring the MNN operator repository based on official documentation and internal operator semantics. As detailed in Appendix B, we dive into the codebase to decouple kernel implementations, ensuring that multiple kernels originally bundled in a single file are refactored into standalone files. This granular decoupling serves two purposes: it facilitates modular operator implementation and streamlines the registration process. The detailed configurations are as follows:

Operator registration. Given that MNN's support for PyTorch operators is still evolving, we utilize ONNX as the intermediate representation for evaluation, adhering to official ONNX operator design principles. Our registration mechanism employs a semantic matching strategy, mapping operator names to their corresponding registration schemas and kernel directory paths within the source tree. During experimentation, we implement a "protected hot-swapping" mechanism: a backup of the original operator implementation is created before replacing it with the generated kernel code. Once the cycle of compilation, verification, and benchmarking is complete, the original implementation is restored, preserving the integrity of the operator repository.

Incremental compilation. We establish our build environment on **Ubuntu 24.04.3 via WSL** using the GCC toolchain. To optimize efficiency, we first pre-compile the entire framework foundation. Following operator registration, we employ an incremental compilation strategy to rebuild the framework and generate the necessary model conversion (**MNNConvert**) and benchmarking tools. For mobile deployment, we utilize the **Android NDK toolchain** to cross-compile the adapted MNN library and binaries for the Android runtime environment.

Correctness verification. To validate functional correctness, we leverage MNN's native model conversion tools. The tool automatically computes the discrepancy between the outputs of the source ONNX model and the converted MNN model under identical input conditions. We enforce a strict numerical tolerance threshold of 1e-4.

On-Device performance benchmarking. Upon passing verification, we conduct performance profiling on the target mobile device. We first prepare the cross-compiled MNN benchmarking binary and the modified model on the host system. These assets are deployed to the device's /data/local/tmp directory via the **Android Debug Bridge (ADB)**. To ensure statistical robustness, we adopt a multi-iteration strategy, executing the inference loop **100 times** to calculate the average latency per operator. Throughout the testing phase, we monitor execution to ensure the device remains in a quiescent state, maintaining a CPU utilization **below 10%** (relative to total capacity of 800% on an 8-core system) to minimize system noise and thermal throttling. Performance metrics are analyzed using MNN's profiling output[4].

This evaluation configuration aims to provide a fair and stable evaluation environment where all operators are measured under consistent device conditions. In real-world mobile workloads, operator execution typically occurs alongside other system activities and model components, where resource contention, memory bandwidth pressure, and thermal effects may reduce the achievable performance. However, accurately reproducing such dynamic system loads in a controlled and reproducible manner is challenging. Similar to common practices in GPU kernel benchmarking, we therefore isolate operator execution and minimize background utilization to approximate the intrinsic computational performance of each operator. In practice, a baseline CPU utilization of around 10% corresponds to the typical system overhead observed on an idle device after boot, providing a reasonable compromise between realism and measurement stability.

[4]https://mnn-docs.readthedocs.io/en/latest/tools/test.html

E GRPO TRAINING DETAILS

In this section, we detail the implementation strategies used during the GRPO training, focusing on reward design, parallel evaluation infrastructure, and cross-platform hardware connectivity.

Reward design. We base our reward formulation on the structure proposed in previous work (Baronio et al., 2025), which incentivizes both correctness and latency reduction. The original baseline reward function is defined as:

$$\text{Reward} = 0.3 \cdot 1_{\{\text{correct}\}} + \frac{T_{\text{baseline}}}{T_{\text{generated}}} \cdot 1_{\{\text{correct}\}} \tag{1}$$

where $1_{\{\text{correct}\}}$ is an indicator that equals 1 if the generated kernel passes numerical verification against the ONNX baseline, and 0 otherwise. T_{baseline} and $T_{\text{generated}}$ represent the inference latency of the baseline and generated kernels, respectively. However, generating C++ kernels for the MNN mobile framework presents distinct challenges compared to CUDA kernel generation. The strict reliance on MNN's internal APIs and memory lifecycle often leads to code that fails to compile initially. Consequently, the model faces the "sparse reward" problem, where the reward remains zero for extended periods, hindering the learning of syntactic and structural correctness. To mitigate this, we introduce an intermediate compilation reward. Our modified reward signal is defined as:

$$\text{Reward} = 0.3 \cdot 1_{\{\text{compile}\}} + 0.3 \cdot 1_{\{\text{correct}\}} + \frac{T_{\text{baseline}}}{T_{\text{generated}}} \cdot 1_{\{\text{correct}\}} \tag{2}$$

By adding the term $0.3 \cdot 1_{\{\text{compile}\}}$, we provide the model with early feedback on syntactical validity, effectively guiding the policy optimization process through the initial cold-start phase.

Parallelized evaluation environment. During the GRPO training phase, the policy generates a group of 5 code samples for each prompt. Since the MNN build system involves complex file dependencies and intermediate object generation, multiple compilation processes cannot share the same working directory without causing race conditions or linkage errors. To enable efficient parallel evaluation, we implement a workspace isolation strategy. For each generated sample in a group, we instantiate an independent working directory by duplicating the necessary MNN core library and build scripts. This allows the compilation and correctness verification of multiple samples to proceed concurrently, significantly accelerating the reward computation without mutual interference.

Remote-mobile bridge connection. Since our training pipeline runs on a remote high-performance Linux server cluster, while the performance measurements must be executed on real-world mobile devices, we established a seamless bridge between the two environments. We use secure shell (SSH) reverse tunneling to forward the local ADB server socket from the host machine to the remote Linux server. This setup allows the training pipeline on the remote server to issue commands such as `adb push` and `adb shell` directly on the mobile device connected to the local host. This architecture supports full bidirectional file transfer and shell execution, enabling the evaluation script to deploy compiled shared libraries to the mobile device and retrieve profiling logs automatically.

F CASE STUDY

As shown in Tab. 6 and Fig. 6, starting from a baseline implementation (1.00x), MoKA demonstrates remarkable efficacy in navigating the complex optimization space, ultimately achieving a peak speedup of 6.82x at epoch 8. The process reveals the agent's ability to systematically identify and address hierarchical bottlenecks. Initially, MoKA targets computational efficiency, introducing SIMD vectorization (epoch 3, 2.06x) and FMA instructions (epoch 5, 4.54x) to maximize instruction throughput. As the compute bound is alleviated, the agent autonomously shifts focus to memory latency. By epoch 8, MoKA successfully implements advanced memory-hiding techniques—specifically 64KB cache blocking with software prefetching—to overcome the memory wall, yielding the global optimal performance. This trajectory confirms that MoKA can effectively synthesize diverse optimization strategies, ranging from instruction-level parallelism to memory hierarchy management, without human intervention.

Performance fluctuations and optimization path diversity. It is worth noting that the optimization curve exhibits a non-monotonic "sawtooth" pattern, particularly the significant performance drops observed at epoch 6 and epoch 9. This phenomenon stems from our design constraint where the

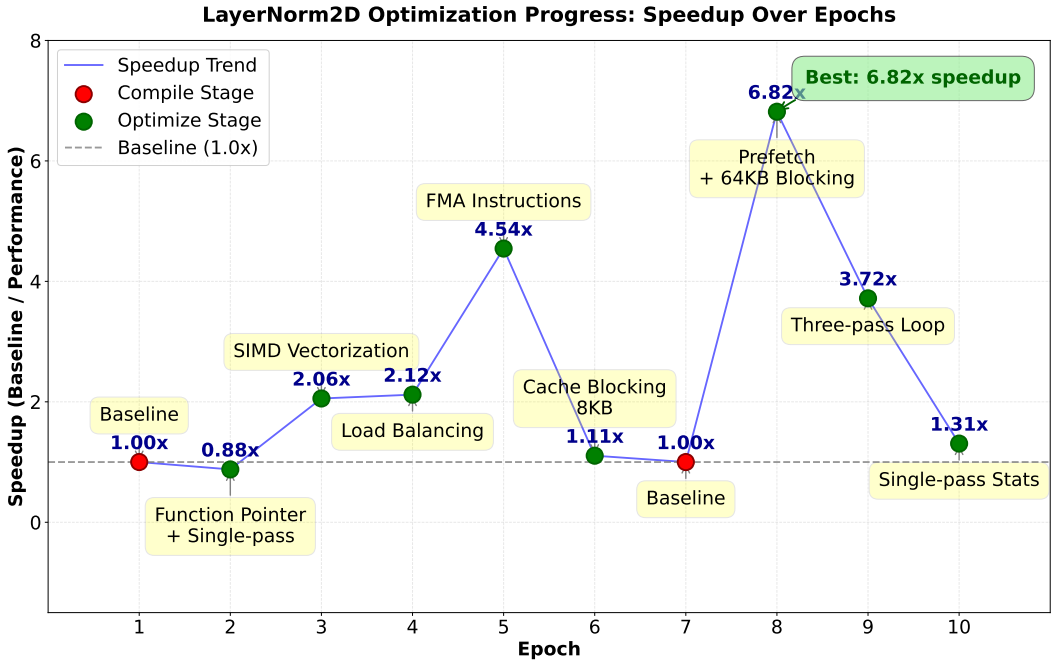


Figure 6: **MoKA optimize LayerNorm2D kernel process.** We conduct the optimization process for 10 iterations. MoKA shows that LLMs can provide diverse optimization methods like SIMD and cache blocking, achieve remarkable speedups.

Table 6: **Case study: LayerNorm2D.** We demonstrate MoKA's capability to automatically optimize kernel implementations through 10 iterative epochs. MoKA achieves a peak speedup of 6.82× at epoch 8, perform an average speedup of 2.82x.

Epoch	Stage	Performance / ms	Baseline / ms	Speedup
1	compilation	–	0.409	–
2	optimization	0..466	0.409	0.88
3	optimization	0.199	0.409	2.06
4	optimization	0.193	0.409	2.12
5	optimization	0.09	0.409	4.54
6	optimization	0.37	0.409	1.11
7	compilation	–	0.409	–
8	optimization	**0.06**	0.409	**6.82**
9	optimization	0.11	0.409	3.72
10	optimization	0.313	0.409	1.31

LLM is prompted to identify and resolve only the single most critical bottleneck in each iteration. This focus forces the agent to choose specific *optimization lanes* that may conflict with previous gains. For instance, the transition from epoch 5 to epoch 6 involved **a shift from a compute-centric strategy (FMA) to a memory-centric strategy (small-block Welford algorithm)**. The overhead introduced by the complex memory access pattern in epoch 6 inadvertently negated the computational gains, causing a temporary regression. However, this volatility is instrumental; the performance drop serves as negative feedback, prompting the agent to self-correct. In subsequent iterations, MoKA refined the blocking strategy and reintroduced prefetching, successfully reconciling the conflict between compute and memory optimizations to reach the global optimum.