QiMeng-Kernel: Macro-Thinking Micro-Coding Paradigm for LLM-Based High-Performance GPU Kernel Generation

Xinguo Zhu[1,3,4], **Shaohui Peng**[1,4*], **Jiaming Guo**[2], **Yunji Chen**[2,4], **Qi Guo**[2], **Yuanbo Wen**[2], **Hang Qin**[1,3,4], **Ruizhi Chen**[1,4], **Qirui Zhou**[2,4], **Ke Gao**[1,4], **Yanjun Wu**[1,4], **Chen Zhao**[1,4], **Ling Li**[1,4*]

[1] Intelligent Software Research Center, Institute of Software, CAS, Beijing, China
[2] State Key Lab of Processors, Institute of Computing Technology, CAS, Beijing, China
[3] Hangzhou Institute for Advanced Study, UCAS, Hangzhou, China
[4] University of Chinese Academy of Sciences
zhuxinguo23@mails.ucas.ac.cn, pengshaohui@iscas.ac.cn, liling@iscas.ac.cn

Abstract

Developing high-performance GPU kernels is critical for AI and scientific computing, but remains challenging due to its reliance on expert crafting and poor portability. While large language models (LLMs) offer promise for automation, both general-purpose and finetuned LLMs suffer from two fundamental and conflicting limitations: correctness and efficiency. The key reason is that existing LLM-based approaches directly generate the entire optimized low-level programs, requiring exploration of an extremely vast space encompassing both optimization policies and implementation codes.

To address the challenge of exploring an intractable space, we propose Macro Thinking Micro Coding (MTMC), a hierarchical framework inspired by the staged optimization strategy of human experts. It decouples optimization strategy from implementation details, ensuring efficiency through high-level strategy and correctness through low-level implementation. Specifically, Macro Thinking employs reinforcement learning to guide lightweight LLMs in efficiently exploring and learning semantic optimization strategies that maximize hardware utilization. Micro Coding leverages general-purpose LLMs to incrementally implement the stepwise optimization proposals from Macro Thinking, avoiding full-kernel generation errors. Together, they effectively navigate the vast optimization space and intricate implementation details, enabling LLMs for high-performance GPU kernel generation.

Comprehensive results on widely adopted benchmarks demonstrate the superior performance of MTMC on GPU kernel generation in both accuracy and running time. On KernelBench, MTMC achieves near 100% and 70% accuracy at Levels 1-2 and 3, over 50% than SOTA general-purpose and domain-finetuned LLMs, with up to 7.3× speedup over LLMs, and 2.2× over expert-optimized PyTorch Eager kernels. On the more challenging TritonBench, MTMC attains up to 59.64% accuracy and 34× speedup. All models and datasets will be made publicly available.

1 Introduction

As the cornerstone of modern artificial intelligence (AI) computing, scientific simulation, and large-scale analytics, GPU kernels face growing demand for high performance, fueled by the quest for computational efficiency across diverse hardware and algorithms (Paszke et al. 2019; Tay et al. 2022; Pandey et al. 2022; Ansel et al. 2024). Yet developing high-performance GPU kernels remains notoriously challenging, heavily reliant on experts' empirical knowledge and hardware insight for the manual tuning process, which is prohibitively expensive and inefficient. For example, FlashAttention implementation on Hopper architecture takes years-long development (Dao et al. 2022; Dao 2023; Shah et al. 2024) with poor portability. Despite the emergence of GPU Domain-Specific Language (DSL) like Triton (Tillet, Kung, and Cox 2019), experts remain essential for designing hardware-specific optimization strategies and data layouts, failing to resolve the inherent complexities or platform portability issues. While established expert-optimized libraries (e.g., generic kernels in PyTorch Eager, as shown in Figure 1(a)) often sacrifice peak performance for specific scenarios, and suffer from limited cross-hardware portability. Consequently, automated generation of high-performance GPU kernels has emerged as a critical and persistent challenge.

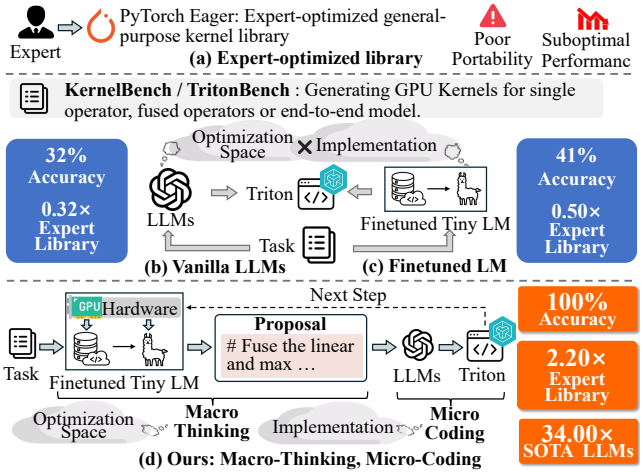

Figure 1: Comparison of GPU kernel generation paradigms.

*Corresponding author.

Given the rapid advances of LLMs in code generation tasks (Le et al. 2022; Joshi et al. 2023; Jiang et al. 2024; Tian et al. 2025), a growing body of research is now focusing on leveraging LLMs to automatically generate high-performance GPU kernels (Zhang et al. 2025; Zhou et al. 2025b). Existing methods employ either powerful general-purpose LLMs or finetuned specialized LLMs, but neither delivers satisfactory performance, as shown in Figure 1(b) and (c). Recent most widely adopted benchmarks (KernelBench (Ouyang et al. 2025), TritonBench(Li et al. 2025)) show that even SOTA general-purpose LLMs struggle to generate correct GPU kernels. They produce significant errors ranging from basic compilation failures to computational flaws across both CUDA and Triton implementations, while performing substantially worse than expert-crafted kernels. Due to the extremely limited availability of optimized kernel datasets (Li et al. 2023), domain-specific finetuned LLMs (e.g., KernelLLM developed by Meta (Fisches et al. 2025), Kevin-32B from Stanford (Baronio et al. 2025)) also suffer from inadequate performance and cross-task generalization. Overall, while LLM-based methods represent a promising direction for automating high-performance GPU kernel generation, they suffer from two fundamental and conflicting limitations: correctness and efficiency.

The core challenge stems from the extreme complexity inherent in the GPU kernel optimization space (already $\approx 10^9$ for one subgraph of neural networks on GPU (Zhai et al. 2024)) coupled with implementation details (hundreds/thousands of code lines) under low-level hardware-specific constraints, which is fundamentally distinct from traditional code generation tasks. Without precise hardware and optimization understanding, LLMs prove incapable of navigating the vast optimization space or managing intricate implementation details, as shown by analysis in KernelBench. High-performance kernels require sophisticated pipeline parallelization and multi-level memory access patterns tailored to specific hardware, necessitating extensive tuning even by experienced experts. Moreover, their complex implementations are vulnerable to minor errors that propagate performance inefficiencies or faults. Thus, prior LLM-based approaches are fundamentally inadequate, generating kernels that are either incorrect or significantly underoptimized.

To address the above core challenge, we innovatively decouple GPU kernel generation into high-level optimization strategy and stepwise optimized code implementation, targeting to enable LLM-based generation of correct, high-performance kernel code with minimal data tuning and human effort, as shown in Figure 1(d). Based on this insight, we propose a novel hierarchical GPU kernel generation framework, Macro Thinking Micro Coding (MTMC). The high-level Macro Thinking progressively generates semantic optimization proposals based on hardware characteristics without implementation details. We utilize reinforcement learning (RL) to drive lightweight LLMs in efficiently learning optimization policies on a compact human-curated dataset to maximize hardware utilization. Simultaneously, the lower-level Micro Coding leverages general-purpose LLMs for stepwise optimization implementation, circumventing errors inherent in whole kernel generation. This decoupled paradigm enables Macro Thinking to eliminate massive finetuning data requirements while empowering Micro Coding to harness LLMs' proficiency in atomic code implementation. Thus, MTMC facilitates the automated generation of correct and high-performance kernels.

The key contributions of this paper are as follows:

- We introduce a novel hierarchical generation paradigm that effectively decouples complex optimization policy design from low-level implementation, enabling LLM to generate high-performance GPU kernels.
- We construct a compact yet effective dataset and environment for efficient kernel optimization policy training, which enables step-by-step guidance for general-purpose LLMs to generate high-performance GPU kernels.
- Results on widely adopted benchmarks show MTMC achieves outstanding performance of GPU kernel generation in both accuracy and running time. On KernelBench, MTMC achieves accuracy of near 100% and 70% at Levels 1-2 and 3 respectively (over 50% than general-purpose and domain-finetuned LLMs) with up to 7.3× speedup over LLMs, and 2.2× over expert-optimized PyTorch Eager kernels. Furthermore, MTMC attains remarkable gains on the more challenging TritonBench, up to 59.64% accuracy and 34 × speedup.

2 Related Works

Benchmarks KernelBench (Ouyang et al. 2025) and TritonBench (Li et al. 2025) are widely adopted GPU kernel generation benchmarks to evaluate the capability of LLMs to generate efficient GPU kernels.

Benchmark	Description	Types
KernelBench	Level 1: 100 single ops	GEMM, Convolution, Softmax, ...
	Level 2: 100 fused ops	GEMM+Max, Conv2d+ReLU, ...
	Level 3: 50 neural networks	LSTM, VGG16, MiniGPT, ViT,...
TritonBench	T: 166 common tasks	Adam, SGD, BatchNorm, Argmax ...
	G: 184 real-world cases	FlashAttention, BMM, Cumsum, ...

Table 1: Details of KernelBench and TritonBench.

LLM-based Kernel Generation The traditional LLM-based code generation focuses on generating functionally correct high-level programming language codes, and has seen rapid development (Sun et al. 2020; Shin and Nam 2021; Li et al. 2022; Chen et al. 2024; Jiang et al. 2024). General-purpose LLMs (like Gemini 2.5 Pro (Comanici et al. 2025) and Claude Sonnet 4 (Anthropic 2025)) and code LLMs (like Qwen-Coder (Hui et al. 2024) and DeepSeek-Coder (Guo et al. 2024; Zhu et al. 2024)) have demonstrated strong capabilities in understanding and generating high-level programming languages across diverse tasks. However, these models struggle with generating correct and high-performance GPU kernels, as they lack comprehensive hardware and optimization understanding. Given the complexity and unique challenges of high-performance kernel gen-

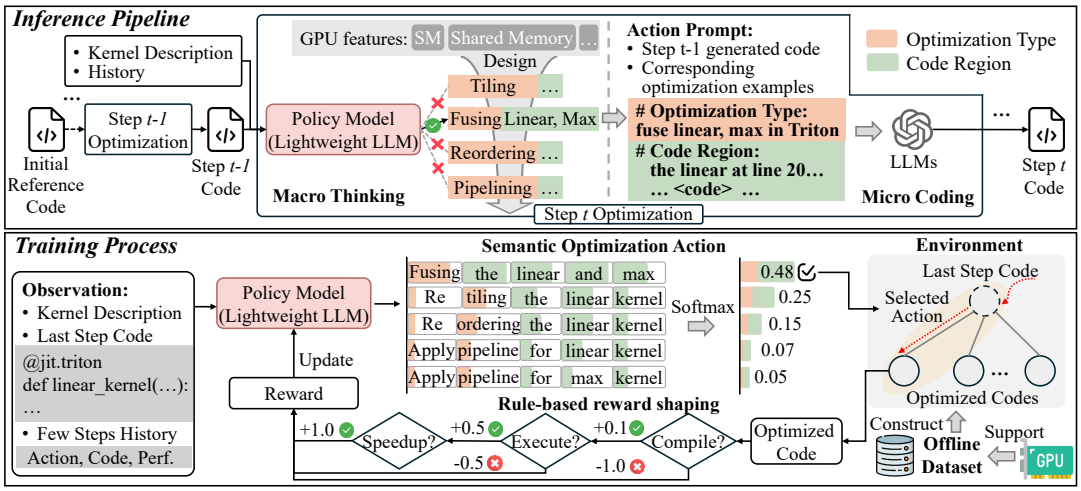

Figure 2: MTMC overview. The framework takes unoptimized PyTorch code as input and generates high-performance GPU kernels with hierarchical process: Macro Thinking generates semantic optimization actions, while Micro Coding implements them step-by-step. The optimization policy based on lightweight LLMs is trained with RL on compact human-craft dataset.

eration, several targeted approaches have emerged. The QiMeng series (Zhou et al. 2025a,b; Zhang et al. 2025; Dong et al. 2025) focuses on leveraging LLMs to generate specific operators through hardware-aware generation and parameter tuning. KernelLLM and Kevin-32B leverage small sets of kernel data to finetune LLMs. Additionally, there are also agent-based approaches like AI CUDA Engineer. However, these works are often constrained by specific operators and limited by data scarcity, resulting in poor generalization and suboptimal performance. Different from previous works, our MTMC decouples high-level strategy from low-level details, achieving efficient learning of general optimization and high correctness.

3 Preliminary

3.1 GPU Kernel Generation Task

GPU kernel generation constitutes a specialized program synthesis task (Manna and Waldinger 1971; Waldinger and Lee 1969), aiming to produce correct and high-performance GPU kernel implementations P from abstract algorithmic specifications S (e.g., natural language descriptions, simple high-level reference code like Python) on a specific GPU h:

$$\begin{aligned} \textbf{minimize}_{P} \quad & \text{RunningTime}(P, h) \\ \textbf{subject to} \quad & P \equiv S \;\&\; \text{Correct}(P, h) = \text{true} \end{aligned} \tag{1}$$

Compared to other code generation tasks, GPU kernel generation presents significantly greater challenges. It necessitates simultaneously satisfying hardware constraints while achieving both functional correctness and high performance. The former requires successful compilation and execution on target hardware with accurate computational outputs; the latter demands maximally leveraging GPUs' complex memory hierarchy and parallel computing units to attain high memory/compute efficiency. This represents a fundamental trade-off: higher-performance optimizations typically entail more intricate implementation details, heightening error susceptibility. Even experts must rely on profound hardware architecture and optimization comprehension, undergoing iterative refinement.

3.2 Optimization Principles

We summarize four fundamental optimization principles corresponding to specific aspects of GPU characteristics to transform a naive algorithm into a complex, high-performance GPU kernel implementation.

- **Tiling** partitions data to fit shared memory size for memory access acceleration (Dally, Keckler, and Kirk 2021).
- **Fusion** merges operators to reduce memory access costs, especially in memory-bound workloads (Shi et al. 2023; Snider and Liang 2023; Wu et al. 2025; Shah et al. 2024).
- **Pipeline** overlaps computation and data movement to improve hardware utilization efficiency (Tan et al. 2011).
- **Reordering** swaps loops to promote memory access locality (Anam, Whatmough, and Andreopoulos 2013).

4 Method

In this section, we introduce the MTMC framework, as in Figure 2, which enables LLMs for high-performance GPU kernel generation by decoupling the optimization and implementation stages to address the efficiency and correctness challenges respectively. The inference pipeline of MTMC and the training process of optimization policy will be detailed in subsequent subsections.

4.1 Inference Pipeline

The Inference Pipeline consists of Macro Thinking and Micro Coding, which decouple the GPU kernel generation into multi-step high-level optimization and corresponding low-level code implementation. Specifically, lightweight LLMs iteratively generate semantic optimization proposals based

on a learned policy to guide general-purpose LLMs in step-by-step kernel generation, in order to ensure maximal correctness while maximizing hardware utilization.

Macro Thinking. It leverages the finetuned lightweight LLMs (training details in Section 4.2) to iteratively generate semantic optimization actions based on hardware information and the code to be optimized. Macro Thinking formulates the optimization process as a stepwise decision-making problem. Each step, the finetuned lightweight LLM policy takes last step kernel code, kernel description and history information as input, then outputs semantic optimization action to Micro Coding stage for implementation, until terminal action or max step is reached. The action declares: (1) the *Optimization Type* (e.g., "fusing"), and (2) the *Code Region* indicating where the optimization should be applied.

Such design enables the policy to generate effective optimization proposals to guide lower-level implementation. First, actions are carefully designed to exploit diverse aspects of GPU hardware features. Second, semantic macro-actions enable efficient policy exploration and learning from offline data on leveraging GPU features performance optimization without concerning of implementation details.

Micro Coding. Micro Coding focuses on translating semantic optimization actions into syntactically correct optimized implementations on kernel. Given the action prompt, Micro Coding output next step kernel code for new round of iteration, by leverages general-purpose LLMs to generate low-level code modifications that implement the intended optimization. As illustrated in Figure 2, the action prompt is automatically constructed by three elements: (1) kernel code of step $t-1$, (2) the semantic optimization action (including type and region) from Macro Thinking, and (3) examples for corresponding optimization type.

Precisely because Macro Thinking demands only atomic, single-step optimizations with explicitly specified type and region, Micro Coding can leverage in-context learning to maximize the probability of generating correct code modifications. With iterative refinement, MTMC ultimately produces correct and high-performance GPU kernels.

4.2 Training Process

To efficiently explore the optimization space and learn an effective optimization policy, we finetune a lightweight LLMs via RL to exploit hardware features. In this section, we detail the training process, focusing on four key aspects: the semantic optimization action space, the training methodology, environment and reward design, as shown in Figure 2 (We provide more training details in the Appendix).

Semantic Optimization Action Space. MTMC enables the policy model to explore and learn kernel optimization strategies by semantically representing optimization actions. Semantic optimization action space are combination of optimization types and the candidate code regions. The optimization types are derived from experts' kernel optimization experience, which focus on leverage different aspects of GPU features. The candidate code regions are determined based on the data flow and abstract syntax tree (AST) analysis to identify syntactically and semantically valid code segment. For example, "fusing the linear and max in line 15 to 20" means fusing adjacent operators to reduce memory access. In summary, the action space refines and extends optimization techniques introduced in Section 3.2, including Tiling, Fusion, reordering, and pipeline. Such action space design is representative for hardware exploitation and can narrow the optimization space, thus supporting Macro Thinking policy efficiently exploring and learning.

Training Methodology. The Macro Thinking policy is a lightweight pre-trained LLM (e.g. DeepSeek-Coder-1.3B and Llama-3.2-1B) , which takes observation as input and then produces semantical optimization actions, as shown in Figure 2. The semantic optimization action $a_k \in \mathcal{A}$ is a sequence of tokens $a_k = \{w_k^1, \ldots, w_k^{N_k}\}$, where N_k is the length of the semantic optimization action k. The action sampling ratio equals to joint probability of all tokens:

$$P_{\text{token}}(a_k|s) = \prod_{i=1}^{N_k} P(w_k^i|s, w_k^1, \ldots, w_k^{i-1}) \tag{2}$$

The sampling probability for each action is then derived by applying Softmax normalization to the logits. We adopt TWOSOME training framework (Tan et al. 2024) and the standard Proximal Policy Optimization (PPO) (Schulman et al. 2017) algorithm to train the policy.

Environment. For representative GPU kernel tasks, including distinct single operations, subgraphs and entire neural networks, we collect a representative **offline dataset** comprising 60k trajectories, without benchmark instances. Each task contains optimization trajectories traversing multiple actions from initial reference code, which provides the foundation for Macro Thinking policy learning. With these offline trajectories, we construct a tree-structured RL environment, as shown in Figure 2. When applying a selected semantic optimization action to current kernel code nodes, the state would transit to varying leaf nodes to provide next step kernel code. This design primarily enables efficient policy training by avoiding prohibitive latency from real-time interaction with LLMs in Micro Coding stage.

Reward Shaping. To guide the learning process, we adopt a rule-based reward shaping across the following criteria. Rewards are assigned based on three criteria from easy to hard: (1) successful compilation, (2) correct executable results, running time without errors, and (3) performance improvement over the previous kernel. Rewards increase progressively while penalties decrease gradually, ensuring the policy initiates exploration from valid optimizations and ultimately learns highly optimized solutions. We further set a step-proportional reward decay mechanism to mitigate degenerate looping behaviors during policy exploration.

5 Results

5.1 Experiment Setup

To study the performance and generality of MTMC, comprehensive evaluations are conducted across 3 different hardware platforms, 13 distinct LLMs and agent, and 2 widely adopted benchmarks (some results refer to Appendix).

LLMs. Marco Thinking is validated with 3 lightweight LLMs, including DeepSeek-Coder-1.3B (Guo et al. 2024), Llama-3.2-1B (Meta 2024), and Qwen2.5-1.5B (Qwen et al.

2025). MTMC defaults to DeepSeek-Coder-1.3B unless specified. Micro Coding is verified on 6 powerful LLMs, including the closed-source models of Gemini 2.5 Pro (Comanici et al. 2025), Gemini 2.5 Flash (Comanici et al. 2025), Claude Sonnet 4 (Anthropic 2025) and OpenAI o4-mini (OpenAI 2025), and the open-source models of DeepSeek-V3 (Liu et al. 2024) and DeepSeek-R1 (Guo et al. 2025).

Hardware. MTMC is tested on three GPUs in Table 2 with the following CPUs: Intel Silver 4114 (V100), Intel Gold 6336Y (A100), and Intel Platinum 8575C (H100).

Feature	V100	A100	H100
Architecture	Volta	Ampere	Hopper
SMs (Streaming Multiprocessors)	80	108	132
Global Memory (GB)	32	80	80
Shared Memory / SM (KB)	96	164	228
L2 Cache (MB)	6	40	50
Memory Bandwidth (GB/s)	900	1935	3350
FP32 TFLOPS	15.7	19.5	60

Table 2: The main features of diverse GPU platforms.

Benchmarks. MTMC is evaluated on KernelBench and TritonBench, as depicted in Table 1. Specifically, KernelBench is tailored to benchmark LLMs in correct and efficient GPU kernel generation across 250 tasks at three levels. Similarly, TritonBench assesses LLM capabilities in Triton kernel generation via 184 real-world kernels (TRITONBENCH-G) and 166 PyTorch-aligned interface kernels (TRITONBENCH-T).

Metrics. Following the benchmark setting and prior work, we adopt four metrics to comprehensively evaluate both correctness and performance of generated kernels. **Call Accuracy** (only adopted in TritonBench) and **Execute Accuracy** measure the rates of kernels that successfully pass compilation and produce correct computation results, respectively. $\textbf{fast}_p$ quantifies the proportion of kernels, which are correct and achieve speedup $>p$ versus PyTorch Eager, across all N tasks. **Mean Speedup** represents the arithmetic mean of speedups. They are calculated as follows:

$$\text{fast}_p = \frac{1}{N}\sum_{i=1}^{N} 1(\text{correct}_i \wedge \{\text{speedup}_i > p\}) \tag{3}$$

$$\text{Mean Speedup} = \frac{1}{N}\sum_{i=1}^{N} \text{speedup}_i \tag{4}$$

Baselines. MTMC undergoes comprehensive comparison against multiple baselines including expert-optimized PyTorch Eager kernels, SOTA general-purpose LLMs (see Table 3 and 4), domain-specific LLMs (Qwen2.5-Coder-32B (Hui et al. 2024)) and agent (Gemini CLI (Google 2025)) for code generation, and intensively finetuned LLMs for GPU kernel generation (Kevin-32B (Baronio et al. 2025) and KernelLLM (Fisches et al. 2025)). See Appendix for details on the baseline setup.

5.2 Overall Performance

Tables 3 and 4 respectively show the results of MTMC across KernelBench and TritonBench, which underscores that MTMC achieves **new SOTA** that empowers LLMs to produce **both correct and high-performing** GPU kernels across diverse hardware and tasks.

KernelBench results demonstrate that MTMC is the **only approach approaching or surpassing expert-optimized kernel performance** in PyTorch Eager (significant $>1\times$ speedup at Levels 1-2 with near-perfect accuracy), dramatically outperforming baselines. Specifically, **versus general-purpose LLMs and code LLMs/agent**: accuracy improves by up to 50% (KernelBench-L2/TritonBench-T) with speedup gains up to 5× (KernelBench-L3). Moreover, **compared to finetuned LLMs** (Kevin-32B/KernelLLM), MTMC delivers $>50\%$ accuracy gains across both benchmarks and achieves 32-34× performance in TritonBench.

MTMC achieves co-optimization of correctness and performance. Defying conventional correctness-performance trade-offs in high-performance kernel generation, MTMC's decoupled generation strategy achieves simultaneous gains. On KernelBench, it delivers dramatic accuracy increase (near 100% accuracy on levels 1 and 2) and up to several-fold improvement in mean speedup (level 3). In contrast, vanilla LLMs generally underperform, while the finetuned LLM (Kevin-32B) significantly sacrifices performance in favor of correctness. On TritonBench, we also achieve SOTA performance and demonstrate substantial gains over the baselines.

MTMC demonstrates strong generalization capabilities across diverse benchmarks and hardware platforms. Despite distinct task characteristics, MTMC achieves significant improvements in correctness and performance over general-purpose LLMs on both benchmarks. Notably, the finetuned LLM (KernelLLM) exhibits severe degradation from KernelBench to TritonBench, accuracy from 40-50% to 2-4% and speedup from $0.5\times$ to $0.01\times$, even performing significantly worse than general-purpose LLMs. This confirms the effectiveness of our decoupled, layered generation mechanism. Furthermore, KernelBench results show our generated kernels deliver consistent performance gains across diverse GPU architectures (V100 to A100, spanning three generations). This suggests our Macro Thinking policy learns universal optimization strategies from limited data.

MTMC maintains robust performance across tasks of varying difficulty levels. MTMC achieves SOTA on varying-difficulty tasks across both benchmarks, significantly surpassing baselines. In contrast, general-purpose/code/finetuned LLMs all show inconsistent results (Tables 1-2, underlined suboptimal entries), due to their exclusive reliance on LLMs' semantic knowledge, causing substantial randomness. MTMC's decoupled design enables robust performance: Macro Thinking learns optimization strategies while Micro Coding generates implementations.

5.3 Ablation Study

Ablation of Target Programming Language: MTMC demonstrates scalability to additional kernel programming languages. While our experiments primarily target Triton due to LLMs' difficulties with low-level CUDA generation (sparse corpus and high complexity), Table 5 reveals MTMC can produce high-performance CUDA kernels with LLMs' greater familiarity operators (like `matmul`). Thus,

Hardware	Method	Level 1			Level 2			Level 3		
		Accuracy(%)	$fast_1$/$fast_2$(%)	Mean Speedup	Accuracy (%)	$fast_1$/$fast_2$(%)	Mean Speedup	Accuracy(%)	$fast_1$/$fast_2$(%)	Mean Speedup
H100	Claude-3.7-Sonnet	32	10 / 2	0.30	11	7 / 0	0.11	12	2 / 0	0.09
	Claude-4-Sonnet	50	26 / 6	1.20	41	32 / 2	0.49	20	6 / 0	0.14
	OpenAI o4-mini	48	23 / 4	1.07	38	25 / 1	0.44	26	6 / 0	0.21
	GPT-4o	20	13 / 3	0.50	2	2 / 0	0.02	6	2 / 0	0.06
	DeepSeek-R1	52	23 / 5	1.18	48	38 / 4	0.65	28	8 / 0	0.23
	DeepSeek-V3-0324	30	14 / 4	0.92	1	1 / 0	0.01	4	2 / 0	0.04
	Llama-3.1-Nemontron	16	10 / 1	0.18	7	5 / 0	0.07	4	2 / 0	0.04
	Qwen3-253B-A22B	49	18 / 4	0.97	39	15 / 2	0.33	10	2 / 0	0.08
	Qwen2.5-Coder-32B	14	8 / 1	0.70	1	1 / 0	0.01	4	0 / 0	0.02
	Gemini CLI	51	32 / 3	1.06	32	25 / 3	0.38	22	14 / 0	0.20
	Kevin-32B (Stanford)	68	9 / 2	0.71	68	24 / 2	0.58	48	4 / 0	0.35
	KernelLLM (Meta)	41	11 / 2	0.38	35	20 / 1	0.41	10	2 / 0	0.09
	Gemini 2.5 Pro	63	31 / 7	1.26	57	34 / 4	0.77	36	6 / 0	0.27
	+ Ours	**100**(↑37%)	**67/13**(↑36%/6%)	**2.08**(↑1.65×)	**99**(↑42%)	**86/12**(↑52%/8%)	**1.28**(↑1.66×)	**70**(↑ 34%)	**34/2**(↑28%/2%)	**0.77**(↑2.85×)
	v.s. Kevin-32B	↑ 32%	↑ 58% / 11%	↑ 2.93×	↑ 31%	↑ 62% / 10%	↑ 2.10×	↑ 22%	↑ 30% / 2%	↑ 8.41 ×
	v.s. KernelLLM	↑ 59%	↑ 56% / 11%	↑ 5.47×	↑ 64%	↑ 66% / 11%	↑ 3.12×	↑ 60%	↑ 32% / 2%	↑ 8.56 ×
	Gemini 2.5 Flash	54	29 / 9	1.25	47	30 / 3	0.53	32	4 / 0	0.15
	+ Ours	**94**(↑40%)	**54/13**(↑ 25%/4%)	**2.03**(↑1.62×)	**97**(↑50%)	**85/5**(↑55%/2%)	**1.35**(↑2.55×)	**64**(↑32%)	**28/2**(↑ 24%/2%)	**0.73**(↑4.87×)
	v.s. Kevin-32B	↑ 26%	↑ 45% / 11%	↑ 2.86×	↑ 29%	↑ 61% / 3%	↑ 2.33×	↑ 16%	↑ 24% / 2%	↑ 2.09×
	v.s. KernelLLM	↑ 53%	↑ 43% / 11%	↑ 5.34×	↑ 62%	↑ 65% / 4%	↑ 3.29×	↑ 54%	↑ 26% / 2%	↑ 8.11×
A100	Claude-3.7-Sonnet	32	8 / 3	0.32	11	4 / 0	0.10	12	6 / 0	0.10
	Claude-4-Sonnet	50	15 / 5	1.31	41	15 / 1	0.39	20	6 / 0	0.15
	OpenAI o4-mini	48	13 / 3	1.27	38	25 / 0	0.39	28	8 / 2	0.26
	GPT-4o	20	8 / 2	0.57	2	2 / 0	0.02	6	6 / 0	0.06
	DeepSeek-R1	52	15 / 4	1.37	46	33 / 1	0.50	28	12 / 0	0.27
	DeepSeek-V3-0324	30	6 / 3	1.12	1	0 / 0	0.01	4	2 / 0	0.04
	Llama-3.1-Nemontron	16	5 / 0	0.16	7	5 / 0	0.07	4	4 / 0	0.04
	Qwen3-253B-A22B	49	8 / 3	1.18	38	12 / 2	0.27	10	2 / 0	0.08
	Qwen2.5-Coder-32B	14	3 / 1	0.94	1	1 / 0	0.01	4	2 / 0	0.02
	Gemini CLI	51	30 / 2	1.29	32	22 / 1	0.34	22	16 / 0	0.20
	Kevin-32B (Stanford)	69	4 / 3	0.84	67	8 / 0	0.40	46	2 / 0	0.32
	KernelLLM (Meta)	41	21 / 2	0.50	35	34 / 2	0.50	10	6 / 0	0.10
	Gemini 2.5 Pro	63	23 / 6	1.47	57	33 / 2	0.69	36	8 / 2	0.14
	+ Ours	**100**(↑37%)	**59/9**(↑36%/3%)	**2.20**(↑1.50×)	**99**(↑42%)	**66/8**(↑33%/6%)	**1.22**(↑1.77×)	**70**(↑34%)	**20/4**(↑12%/2%)	**0.73**(↑5.21×)
	v.s. Kevin-32B	↑ 31%	↑ 55% / 6%	↑ 2.62×	↑ 32%	↑ 58% / 8%	↑ 3.05×	↑ 24%	↑ 18% / 4%	↑ 2.28×
	v.s. KernelLLM	↑ 59%	↑ 38% / 7%	↑ 4.40×	↑ 64%	↑ 32% / 6%	↑ 2.44×	↑ 60%	↑ 14% / 4%	↑ 7.30×
	Gemini 2.5 Flash	54	21 / 7	1.42	47	30 / 2	0.51	32	4 / 0	0.12
	+ Ours	**94**(↑40%)	**53/10**(↑32%/3%)	**2.14**(↑1.51×)	**97**(↑50%)	**56/4**(↑26%/2%)	**1.21**(↑2.37×)	**64**(↑32%)	**48/2**(↑44%/2%)	**0.69**(↑5.75×)
	v.s. Kevin-32B	↑ 25%	↑ 49% / 7%	↑ 2.55×	↑ 30%	↑ 48% / 4%	↑ 3.03×	↑ 18%	↑ 46% / 2%	↑ 2.16×
	v.s. KernelLLM	↑ 53%	↑ 32% / 8%	↑ 4.28×	↑ 62%	↑ 22% / 2%	↑ 2.42×	↑ 54%	↑ 42% / 2%	↑ 6.90×
V100	Claude-3.7-Sonnet	32	7 / 2	0.33	11	2 / 0	0.10	6	4 / 0	0.06
	Claude-4-Sonnet	50	17 / 5	1.17	41	29 / 0	0.46	20	0 / 0	0.14
	OpenAI o4-mini	48	15 / 3	1.06	38	25 / 1	0.40	26	2 / 0	0.22
	GPT-4o	20	9 / 2	0.63	2	0 / 0	0.02	6	0 / 0	0.06
	DeepSeek-R1	51	14 / 4	1.13	48	33 / 3	0.56	26	10 / 0	0.24
	DeepSeek-V3-0324	30	7 / 3	0.93	1	0 / 0	0.01	4	2 / 0	0.04
	Llama-3.1-Nemontron	16	6 / 0	0.17	7	5 / 0	0.07	4	2 / 0	0.04
	Qwen3-253B-A22B	48	10 / 3	1.02	38	12 / 3	0.28	10	0 / 0	0.09
	Qwen2.5-Coder-32B	14	4 / 1	0.68	1	0 / 0	0.01	2	0 / 0	0.02
	Gemini CLI	49	16 / 3	1.07	32	22 / 1	0.35	22	8 / 0	0.20
	Kevin-32B (Stanford)	69	6 / 2	0.86	67	14 / 0	0.48	46	4 / 0	0.35
	KernelLLM (Meta)	41	14 / 2	0.52	35	31 / 2	0.49	10	4 / 0	0.10
	Gemini 2.5 Pro	63	17 / 6	1.27	57	29 / 2	0.65	36	8 / 0	0.27
	+ Ours	**99**(↑36%)	**43/9**(↑26%/3%)	**1.94**(↑1.53×)	**100**(↑43%)	**51/3**(↑22%/1%)	**1.04**(↑1.60×)	**70**(↑34%)	**24/2**(↑16%/2%)	**0.65**(↑2.41×)
	v.s. Kevin-32B	↑ 30%	↑ 37% / 7%	↑ 2.26×	↑ 33%	↑ 37% / 3%	↑ 2.17×	↑ 24%	↑ 20% / 2%	↑ 1.86×
	v.s. KernelLLM	↑ 58%	↑ 29% / 7%	↑ 3.73×	↑ 65%	↑ 20% / 1%	↑ 2.12×	↑ 60%	↑ 20% / 2%	↑ 6.50×
	Gemini 2.5 Flash	53	16 / 8	1.15	47	27 / 1	0.45	32	4 / 0	0.15
	+ Ours	**93**(↑40%)	**45/11**(↑29%/3%)	**1.93**(↑1.68×)	**97**(↑50%)	**56/3**(↑29%/2%)	**1.18**(↑2.62×)	**64**(↑32%)	**36/2**(↑32%/2%)	**0.68**(↑4.53×)
	v.s. Kevin-32B	↑ 24%	↑ 39% / 9%	↑ 2.24×	↑ 30%	↑ 42% / 3%	↑ 2.46×	↑ 18%	↑ 32% / 2%	↑ 1.94×
	v.s. KernelLLM	↑ 52%	↑ 31% / 9%	↑ 3.71×	↑ 62%	↑ 25% / 1%	↑ 2.41×	↑ 54%	↑ 32% / 2%	↑ 6.80×

Table 3: The execute accuracy and mean speedup relative to PyTorch Eager for Triton kernel generation on KernelBench across various hardware, LLMs and agent. The best results are highlighted in **bold**, while the second-best results are underlined.

Method	TRITONBENCH-G				TRITONBENCH-T			
	Call Accuracy(%)	Execute Accuracy(%)	$fast_1$/$fast_2$(%)	Mean Speedup	Call Accuracy(%)	Execute Accuracy(%)	$fast_1$/$fast_2$(%)	Mean Speedup
Gemini 2.5 Pro	25.00	16.00	9.24 / 1.09	0.28	28.92	22.89	6.63 / 1.81	0.24
Claude-3.7-Sonnet	11.41	8.70	3.80 / 0.54	0.17	14.46	9.64	2.41 / 0.60	0.08
Claude-4-Sonnet	25.54	17.39	7.07 / 0.54	0.27	16.27	10.84	3.01 / 1.20	0.10
OpenAI o4-mini	11.96	7.07	3.26 / 1.09	0.18	8.43	7.83	1.81 / 0.60	0.09
GPT-4o	5.98	3.80	1.63 / 0.54	0.13	6.63	2.41	1.20 / 0.00	0.03
DeepSeek-R1-0528	15.76	8.70	5.98 / 1.09	0.23	28.31	17.47	6.02 / 1.20	0.20
DeepSeek-V3-0324	11.41	7.61	3.26 / 0.54	0.17	18.67	13.86	4.82 / 0.60	0.15
Qwen2.5-Coder-32B	4.35	3.26	1.09 / 1.09	0.08	4.82	3.01	0.60 / 0.00	0.03
KernelLLM (Meta)	2.17	1.09	0.54 / 0.00	0.01	4.82	4.22	0.60 / 0.00	0.02
Gemini 2.5 Flash	11.41	8.70	4.35 / 1.63	0.24	14.46	9.04	5.42 / 1.81	0.15
+ Ours	**32.61**	**22.83**	**9.78 / 1.63**	**0.34**	**64.46**	**54.82**	**19.28 / 3.01**	**0.64**
	↑ 21.2%	↑ 14.13%	↑ 5.41% / -	↑ 1.42×	↑ 50.00%	↑ 45.78%	↑ 13.86% / 1.20%	↑ 4.67×
v.s. KernelLLM	↑ 30.44%	↑ 21.74%	↑ 9.24% / 1.63%	↑ 34.00×	↑ 59.64%	↑ 50.60%	↑ 19.22% / 3.01%	↑ 32.00×

Table 4: Performance comparison across LLMs and agents for Triton kernel generation on TritonBench with A100 GPU.

Task ID	1	2	6	7	8	9	13
MTMC (Triton)	1.38	1.36	4.43	1.45	37.88	0.92	10.56
MTMC (CUDA)	1.38	1.66	1.34	1.66	26.52	0.91	10.17

Table 5: Execution time (ms) of MTMC on KernelBench `matmul` operators with different generation targets.

the primary scalability bottleneck resides in the LLM's proficiency with the target programming language.

Ablation of Micro Coding: Hierarchical kernel generation with multi-step implementation is critical. As shown in Table 6, feeding all optimization actions and corresponding prompts at once to the LLM (w/o Hier) causes significant performance degradation (up to 64% accuracy and 1.3× speedup). This indicates that current SOTA LLMs cannot generate such complex kernels in a single pass, confirming the rationale behind our hierarchical step-by-step optimization and implementation design.

KernelBench Level	Level 1	Level 2	Level 3
	Accuracy/Speedup	Accuracy/Speedup	Accuracy/Speedup
GF-2.5 w/o Hier	60% / 1.38	32% / 0.43	10% / 0.09
GF-2.5 + Ours	**94% / 2.14**	**97% / 1.21**	**64% / 0.69**
DS-V3 w/o Hier	41% / 0.52	16% / 0.17	6% / 0.04
DS-V3 + Ours	**78% / 1.82**	**59% / 0.66**	**36% / 0.32**

Table 6: Comparison between multi-step (ours) and single-pass generation (w/o Hier).

Ablation of Macro Thinking: Table 7 presents ablation on two key designs of the Macro Thinking component:"policy" indicating whether to learn optimization policies, and "AS" determining whether to leverage action spaces based on hardware-aware optimization summary. **(1) The efficacy of the Macro Thinking policy exhibits robustness to the choice of lightweight LLMs.** Policies with diverse LLMs consistently achieve high correctness and speedup, with the smallest model paradoxically delivering the best results. This demonstrates the exceptional efficiency of our policy training paradigm. **(2) Reinforcement learning for optimization policies is essential**. Directly employing LLMs for Macro Thinking results in a marked performance decrease, despite task simplification through decoupling. **(3) The action space design proves both rational and critical.** Without this, unrestricted LLM-generated suggestions cause further performance degradation. This finding confirms LLMs' fundamental lack of comprehension regarding hardware-aware optimizations.

Setting	Method	Level 1	Level 2	Level 3
w/ policy w/ AS	- DS-Coder	90% / 1.10	100% / 1.16	100% / 1.82
	- Llama	100% / 1.17	80% / 0.86	80% / 0.74
	- Qwen	90% / 0.97	100% / 0.90	100% / 0.75
w/o policy w/ AS	- random	70% / 0.50	50% / 0.51	40% / 0.15
	- GPT-4o	50% / 0.74	40% / 0.51	40% / 0.29
	- DS-V3	50% / 0.67	50% / 0.60	60% / 0.61
	- GF-2.5	50% / 0.82	60% / 0.62	60% / 0.53
w/o policy w/o AS	- GPT-4o	20% / 0.16	30% / 0.28	20% / 0.03
	- DS-V3	30% / 0.14	10% / 0.02	20% / 0.34
	- GF-2.5	30% / 0.25	50% / 0.49	40% / 0.36

Table 7: The execute accuracy and mean speedup comparison of different policies and settings. 10% of KernelBench tasks are used for testing.

There are also two ablations in the Appendix: one on Micro Coding models that further demonstrates MTMC's generalization across LLMs, and another indicating that MTMC reaches peak performance with only a few optimization steps, while LLMs cannot promote through resampling.

6 Conclusion

In this paper, we propose MTMC, a hierarchical framework that decouples high-level optimization strategy and low-level implementation and enables LLMs to automatically generate correct and high-performance GPU kernels.

Results on widely adopted benchmarks show MTMC significantly outperforms existing LLMs in both SOTA correctness and performance, and is the only method significantly surpassing expert-optimized PyTorch Eager kernels in most tasks. MTMC currently focuses on GPU / Triton and can be further improved in the network-level kernel (already significantly surpassing other works). We will extend it to more emerging hardware platforms and languages, and further promote its ability in future work.

Acknowledgments

This work is partially supported by the NSF of China (under Grant 92364202), and Major Program of ISCAS (Grant No. ISCAS-ZD-202402).

References

Anam, M. A.; Whatmough, P. N.; and Andreopoulos, Y. 2013. Precision-energy-throughput scaling of generic matrix multiplication and discrete convolution kernels via linear projections. In *The 11th IEEE Symposium on Embedded Systems for Real-time Multimedia*, 21–30. IEEE.

Ansel, J.; Yang, E.; He, H.; Gimelshein, N.; Jain, A.; Voznesensky, M.; Bao, B.; Bell, P.; Berard, D.; Burovski, E.; et al. 2024. 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*, 929–947.

Anthropic. 2025. Introducing Claude 4. https://www.anthropic.com/news/claude-4. Accessed: 2025-07-30.

Baronio, C.; Marsella, P.; Pan, B.; Guo, S.; and Alberti, S. 2025. Kevin: Multi-turn rl for generating cuda kernels. *arXiv preprint arXiv:2507.11948*.

Chen, J.; Tang, H.; Chu, Z.; Chen, Q.; Wang, Z.; Liu, M.; and Qin, B. 2024. Divide-and-conquer meets consensus: Unleashing the power of functions in code generation. *Advances in Neural Information Processing Systems*, 37: 67061–67105.

Comanici, G.; Bieber, E.; Schaekermann, M.; Pasupat, I.; Sachdeva, N.; Dhillon, I.; Blistein, M.; Ram, O.; Zhang, D.; Rosen, E.; et al. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*.

Dally, W. J.; Keckler, S. W.; and Kirk, D. B. 2021. Evolution of the graphics processing unit (GPU). *IEEE Micro*, 41(6): 42–51.

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

Dao, T.; Fu, D.; Ermon, S.; Rudra, A.; and Ré, C. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35: 16344–16359.

Dong, S.; Wen, Y.; Bi, J.; Huang, D.; Guo, J.; Xu, J.; Xu, R.; Song, X.; Hao, Y.; Zhou, X.; et al. 2025. QiMeng-Xpiler: Transcompiling tensor programs for deep learning systems with a neural-symbolic approach. *arXiv preprint arXiv:2505.02146*.

Fisches, Z. V.; Paliskara, S.; Guo, S.; Zhang, A.; Spisak, J.; Cummins, C.; Leather, H.; Synnaeve, G.; Isaacson, J.; Markosyan, A.; and Saroufim, M. 2025. KernelLLM: Making Kernel Development More Accessible. Corresponding authors: Aram Markosyan, Mark Saroufim.

Google. 2025. Gemini CLI: your open-source AI agent. https://blog.google/technology/developers/introducing-gemini-cli-open-source-ai-agent/. Accessed: 2025-07-30.

Guo, D.; Yang, D.; Zhang, H.; Song, J.; Zhang, R.; Xu, R.; Zhu, Q.; Ma, S.; Wang, P.; Bi, X.; et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Guo, D.; Zhu, Q.; Yang, D.; Xie, Z.; Dong, K.; Zhang, W.; Chen, G.; Bi, X.; Wu, Y.; Li, Y.; et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196*.

Hui, B.; Yang, J.; Cui, Z.; Yang, J.; Liu, D.; Zhang, L.; Liu, T.; Zhang, J.; Yu, B.; Lu, K.; et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Jiang, J.; Wang, F.; Shen, J.; Kim, S.; and Kim, S. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.

Joshi, H.; Sanchez, J. C.; Gulwani, S.; Le, V.; Verbruggen, G.; and Radiček, I. 2023. Repair is nearly generation: Multilingual program repair with llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, 5131–5140.

Le, H.; Wang, Y.; Gotmare, A. D.; Savarese, S.; and Hoi, S. C. H. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35: 21314–21328.

Li, J.; Li, S.; Gao, Z.; Shi, Q.; Li, Y.; Wang, Z.; Huang, J.; Wang, H.; Wang, J.; Han, X.; et al. 2025. TritonBench: Benchmarking Large Language Model Capabilities for Generating Triton Operators. *arXiv preprint arXiv:2502.14752*.

Li, R.; Allal, L. B.; Zi, Y.; Muennighoff, N.; Kocetkov, D.; Mou, C.; Marone, M.; Akiki, C.; Li, J.; Chim, J.; et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Dal Lago, A.; et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624): 1092–1097.

Liu, A.; Feng, B.; Xue, B.; Wang, B.; Wu, B.; Lu, C.; Zhao, C.; Deng, C.; Zhang, C.; Ruan, C.; et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Manna, Z.; and Waldinger, R. J. 1971. Toward automatic program synthesis. *Communications of the ACM*, 14(3): 151–165.

Meta. 2024. Llama 3.2: Revolutionizing edge AI and vision with open, customizable models. https://ai.meta.com/blog/llama-3-2-connect-2024-vision-edge-mobile-devices/. Accessed: 2025-07-30.

OpenAI. 2025. Introducing OpenAI o3 and o4-mini. https://openai.com/index/introducing-o3-and-o4-mini. Accessed: 2025-07-30.

Ouyang, A.; Guo, S.; Arora, S.; Zhang, A. L.; Hu, W.; Ré, C.; and Mirhoseini, A. 2025. Kernelbench: Can llms write efficient gpu kernels? *arXiv preprint arXiv:2502.10517*.

Pandey, M.; Fernandez, M.; Gentile, F.; Isayev, O.; Tropsha, A.; Stern, A. C.; and Cherkasov, A. 2022. The transformational role of GPU computing and deep learning in drug discovery. *Nature Machine Intelligence*, 4(3): 211–221.

Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32.

Qwen; :; Yang, A.; Yang, B.; Zhang, B.; Hui, B.; Zheng, B.; Yu, B.; Li, C.; Liu, D.; Huang, F.; Wei, H.; Lin, H.; Yang, J.; Tu, J.; Zhang, J.; Yang, J.; Yang, J.; Zhou, J.; Lin, J.; Dang, K.; Lu, K.; Bao, K.; Yang, K.; Yu, L.; Li, M.; Xue, M.; Zhang, P.; Zhu, Q.; Men, R.; Lin, R.; Li, T.; Tang, T.; Xia, T.; Ren, X.; Ren, X.; Fan, Y.; Su, Y.; Zhang, Y.; Wan, Y.; Liu, Y.; Cui, Z.; Zhang, Z.; and Qiu, Z. 2025. Qwen2.5 Technical Report. arXiv:2412.15115.

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Shah, J.; Bikshandi, G.; Zhang, Y.; Thakkar, V.; Ramani, P.; and Dao, T. 2024. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *Advances in Neural Information Processing Systems*, 37: 68658–68685.

Shi, Y.; Yang, Z.; Xue, J.; Ma, L.; Xia, Y.; Miao, Z.; Guo, Y.; Yang, F.; and Zhou, L. 2023. Welder: Scheduling deep learning memory access via tile-graph. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 701–718.

Shin, J.; and Nam, J. 2021. A survey of automatic code generation from natural language. *Journal of Information Processing Systems*, 17: 537–555.

Snider, D.; and Liang, R. 2023. Operator fusion in XLA: analysis and evaluation. *arXiv preprint arXiv:2301.13062*.

Sun, Z.; Zhu, Q.; Xiong, Y.; Sun, Y.; Mou, L.; and Zhang, L. 2020. Treegen: A tree-based transformer architecture for code generation. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, 8984–8991.

Tan, G.; Li, L.; Triechle, S.; Phillips, E.; Bao, Y.; and Sun, N. 2011. Fast implementation of DGEMM on Fermi GPU. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–11.

Tan, W.; Zhang, W.; Liu, S.; Zheng, L.; Wang, X.; and An, B. 2024. True knowledge comes from practice: Aligning llms with embodied environments via reinforcement learning. *arXiv preprint arXiv:2401.14151*.

Tay, Y.; Dehghani, M.; Bahri, D.; and Metzler, D. 2022. Efficient Transformers: A Survey. *ACM Comput. Surv.*, 55(6).

Tian, Y.; Yan, W.; Yang, Q.; Zhao, X.; Chen, Q.; Wang, W.; Luo, Z.; Ma, L.; and Song, D. 2025. Codehalu: Investigating code hallucinations in llms via execution-based verification. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, 25300–25308.

Tillet, P.; Kung, H.-T.; and Cox, D. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 10–19.

Waldinger, R. J.; and Lee, R. C. 1969. PROW: A step toward automatic program writing. In *Proceedings of the 1st international joint conference on Artificial intelligence*, 241–252.

Wu, M.; Cheng, X.; Liu, S.; Shi, C.; Ji, J.; Ao, M. K.; Velliengiri, P.; Miao, X.; Padon, O.; and Jia, Z. 2025. Mirage: A Multi-Level Superoptimizer for Tensor Programs. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, 21–38.

Zhai, Y.; Yang, S.; Pan, K.; Zhang, R.; Liu, S.; Liu, C.; Ye, Z.; Ji, J.; Zhao, J.; Zhang, Y.; et al. 2024. Enabling Tensor Language Model to Assist in Generating {High-Performance} Tensor Programs for Deep Learning. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 289–305.

Zhang, X.; Peng, S.; Zhou, Q.; Wen, Y.; Guo, Q.; Chen, R.; Zhu, X.; Xiong, W.; Chen, H.; Ma, C.; et al. 2025. Qimeng-tensorop: Automatically generating high-performance tensor operators with hardware primitives. *arXiv preprint arXiv:2505.06302*.

Zhou, Q.; Peng, S.; Xiong, W.; Chen, H.; Wen, Y.; Li, H.; Li, L.; Guo, Q.; Zhao, Y.; Gao, K.; et al. 2025a. QiMeng-Attention: SOTA Attention Operator is generated by SOTA Attention Algorithm. *arXiv preprint arXiv:2506.12355*.

Zhou, Q.; Wen, Y.; Chen, R.; Gao, K.; Xiong, W.; Li, L.; Guo, Q.; Wu, Y.; and Chen, Y. 2025b. QiMeng-GEMM: Automatically Generating High-Performance Matrix Multiplication Code by Exploiting Large Language Models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, 22982–22990.

Zhu, Q.; Guo, D.; Shao, Z.; Yang, D.; Wang, P.; Xu, R.; Wu, Y.; Li, Y.; Gao, H.; Ma, S.; et al. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*.