



TensorAgent: Multi-agent Framework for Automated Tensor Core Code Generation

Bingjie Liu, Zhongchun Zheng, and Xianwei Zhang^(✉)

School of Computer Science and Engineering, Sun Yat-sen University,
Guangzhou, China

{liubj8,zhengzhch3}@mail2.sysu.edu.cn, zhangxw79@mail.sysu.edu.cn

Abstract. WMMA (Warp Matrix Multiply-Accumulate) is the CUDA warp-level interface for matrix multiply-accumulate on Tensor Cores, making it a central building block of high-throughput deep learning training and inference. In practice, developing WMMA kernels remains difficult because API-level correctness, fragment memory and layout constraints, and runtime numerical behavior under precision-specific rules must be satisfied jointly. LLM-based CUDA kernel generation has progressed quickly, yet existing methods offer limited support for structured, failure-aware repair across development stages and thus leave much of the potential of LLMs for WMMA-focused Tensor Core programming unrealized. We present TensorAgent, a multi-agent framework that separates planning, synthesis, and execution diagnosis and connects them with hierarchical feedback as an explicit repair policy. That feedback is directed at failures arising during algorithm design, compilation, and execution. We evaluate on 15 WMMA benchmarks spanning six data-type settings and two kernel-complexity tiers and report execution time relative to expert-written baselines. Relative to direct prompting, TensorAgent improves run pass@1 by up to 43.2% points and reaches 99.0% run pass@1 on simple kernels.

Keywords: Multi-agent systems · Code generation · Tensor Core · CUDA · Large language models

1 Introduction

The rapid scaling of deep learning has driven extraordinary demand for GPU compute. Training and inference throughput hinges on a few compute-intensive operators—matrix multiply, attention, convolution, and element-wise fusions—whose efficiency dictates end-to-end performance. On modern NVIDIA GPUs, Tensor Cores provide the primary dense-arithmetic throughput for these operators.

The standard warp-level programming interface for Tensor Cores is WMMA (warp matrix multiply-accumulate) [1]. WMMA coordinates 32 threads within

a warp to load, compute, and store matrix fragments across multiple precision formats (FP16, BF16, TF32, FP64, INT8). Despite its central role, writing correct WMMA kernels remains difficult: developers must simultaneously satisfy API-level constraints (fragment types, supported operations), memory and layout rules, and runtime numerical correctness. The resulting error space is layered: a kernel may compile yet violate layout constraints at runtime, or pass layout checks yet produce numerically wrong results.

Large language models trained on code generate high-quality programs from natural language [2–4], and LLM-based GPU kernel generation is advancing rapidly [5, 6]. However, recent benchmarks reveal a persistent gap between compilation success and functional correctness [6]: code that compiles is far from code that runs correctly. Pass@1 is the fraction of independent trials in which the first generated sample meets a correctness criterion; we report compile pass@1 and run pass@1 separately. Figure 1 summarizes this setting: the left panel shows that direct prompting is largely feed-forward with no explicit error feedback from later stages, while the right panel shows the substantial compile–run gap that results. Existing approaches provide limited support for structured, failure-aware repair across algorithm design, compilation, and runtime, leaving the potential of LLMs for WMMA-oriented Tensor Core programming underused. Multi-agent coding systems show that explicit roles and feedback loops improve complex synthesis [7–10], yet their task-generic decompositions do not match the layered error structure of Tensor Core development. We therefore propose TensorAgent, a multi-agent framework in which each agent aligns with a distinct constraint layer—planning, code synthesis, and execution diagnosis—linked by hierarchical feedback that routes each failure to the lowest level capable of resolving it.

Our contributions are threefold. First, we propose TensorAgent, a multi-agent framework that maps planning, code generation, and execution diagnosis onto the WMMA failure stratification (algorithm design, compilation, and runtime) and links them with hierarchical feedback that routes each failure to the lowest layer able to resolve it before escalating. Second, we construct a benchmark of 15 WMMA kernel tasks in simple and complex layers, spanning BF16, FP16, FP64, INT8, TF32, and mixed-precision workloads. Third, we evaluate three LLMs with 100 trials per kernel per method (9,000 runs): TensorAgent consistently improves compile and run pass@1 over direct prompting, and we report the same metrics under CudaForge [11], a state-of-the-art agentic CUDA workflow with hardware feedback, as a workflow-level reference (Sect. 5.3). We additionally report execution time relative to expert baselines (Sect. 5.4) and practical-cost statistics—repair iterations, plan-escalation rate, and wall-clock time (Sect. 5.6).

2 Related Work

LLM Code Generation and Iterative Prompting. Large language models trained on code achieve strong results on standard benchmarks, from Codex [2] to open models such as StarCoder and Code Llama [3, 4]. Reasoning-oriented methods such as chain-of-thought [12], self-debugging [13], and flow engineering [14]

improve correctness by introducing intermediate structure. Most evaluations, however, focus on algorithmic benchmarks where correctness is test-driven and platform details are abstracted away. Tensor Core kernel generation differs: validity depends on low-level API rules and hardware runtime behavior that standard benchmarks rarely capture.

Multi-agent Coding Systems. Prior multi-agent coding systems (AgentCoder [7], MapCoder [8], CodeAgent [9], CODESIM [10]) demonstrate that role decomposition and iterative collaboration improve complex coding outcomes. These systems are nevertheless largely task-generic and are typically evaluated on repository-level or algorithmic tasks; they do not explicitly model the compile-run failure stratification of Tensor Core development. Our setting requires a domain-specific decomposition in which each role aligns with a distinct constraint layer: algorithmic design, API-correct synthesis, and runtime verification.

GPU Kernel Stacks and Tensor Core Programming. CUDA Agent [15] scales agentic CUDA generation with reinforcement learning, emphasizing training infrastructure rather than lightweight inference-time escalation across design, compilation, and runtime. CudaForge [11] represents the state of the art among *agentic* CUDA workflows, integrating hardware feedback for iterative optimization. We adopt CudaForge as the primary workflow reference on the same WMMA template tasks, so each model is evaluated under both single-shot prompting and a hardware-feedback pipeline. CudaForge emphasizes hardware-guided optimization loops, whereas TensorAgent emphasizes robust WMMA synthesis from fixed templates by localizing failures hierarchically across algorithm design, compilation, and runtime. Broader surveys [16,17] identify specialization and feedback as central mechanisms in LLM multi-agent systems; we apply these mechanisms to WMMA-oriented Tensor Core generation and evaluate them through kernel-level compile and run outcomes.

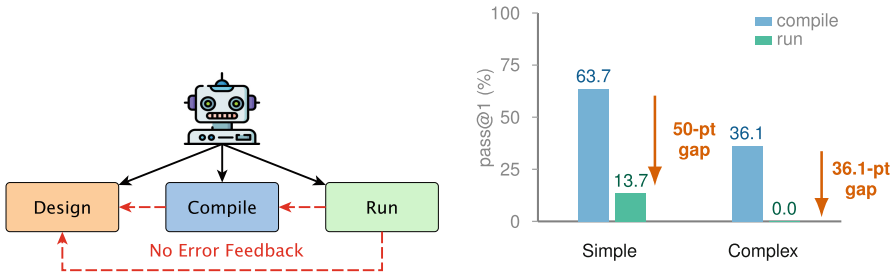


Fig. 1. Direct prompting lacks error feedback from compilation and execution stages (left), resulting in low run pass@1 despite moderate compile success (right).

3 Motivation

Preliminary baseline experiments on our WMMA benchmark (direct single-shot prompting; details in Sect. 5.3) reveal two patterns, previewed in Fig. 1, that directly motivate the architectural design choices in TensorAgent.

Observation 1: Compile Success Does Not Imply Runtime Correctness. Under direct prompting, the compile-to-run gap is large. For example, DeepSeek-v3.2 compiles 63.7% of simple-layer trials yet only 13.7% run correctly—a 50-point gap; on complex kernels, it compiles 36.1% but achieves 0% runtime success (Table 4). The pattern holds across models: many WMMA faults (layout mismatches, synchronization errors, precision-dependent accumulation bugs) pass `nvcc` yet surface only at execution. This gap, documented also in broader GPU benchmarks [6], shows that any effective workflow must include explicit runtime verification rather than treating compilation as a sufficient correctness signal.

Observation 2: Failures Span Algorithm, API, and Runtime Levels. Baseline failures distribute across three levels: algorithm-design errors (wrong tiling or layout), API misuse caught by the compiler (incorrect API calls), and runtime-only faults (synchronization races, numerical accumulation errors). A single undifferentiated feedback channel mixes these signals, increasing cognitive load [18] and leading to repeated surface-level patches that never address the underlying design flaw.

Together, these observations motivate a multi-agent framework that aligns each agent with a distinct failure level—algorithm design, code synthesis, and execution diagnosis—and connects them through hierarchical feedback so that each error is routed to the agent best positioned to resolve it with minimal cross-stage interference.

4 Framework Design

4.1 Overview

Given a CUDA source file whose kernel body has been replaced by a placeholder, TensorAgent produces a correct kernel implementation through the coordinated action of three specialized agents (Fig. 2). PLANAGENT analyzes the template and emits a structured plan fixing algorithm-level decisions before any code is written. GENAGENT converts the plan into compilable CUDA and iteratively repairs compiler errors through a local self-feedback loop with `nvcc`. EXECAGENT runs the compiled binary on the target GPU, and when execution fails, produces a structured diagnostic routed to GENAGENT or escalated to PLANAGENT depending on fault severity.

The key design principle is hierarchical feedback: each error is first handled at the lowest layer capable of resolving it, and escalation occurs only when local repair is insufficient.

The remainder of this section describes the feedback mechanism (Sect. 4.2) and then each agent in detail (Sects. 4.3–4.5).

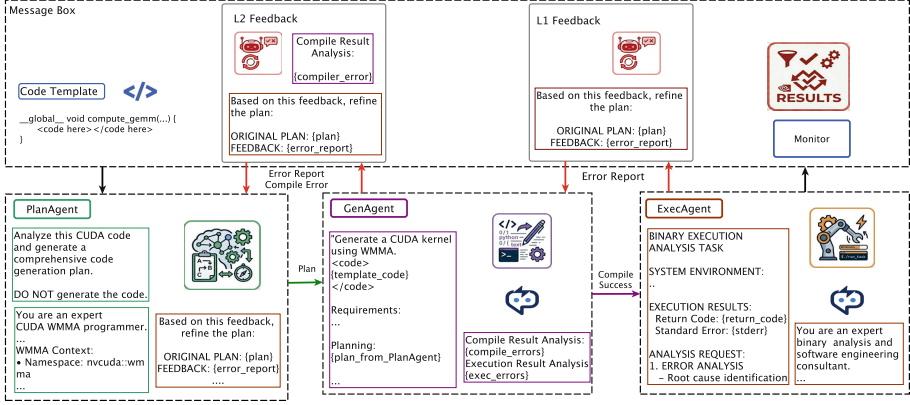


Fig. 2. Overview of TensorAgent. Arrows indicate feedback flow. GENAGENT maintains a local compile-feedback loop; execution failures are diagnosed by EXECAGENT and routed back to GENAGENT. Unresolved errors escalate to PLANAGENT for plan revision.

4.2 Hierarchical Feedback

TensorAgent is organized around *hierarchical feedback* (Sect. 3), which routes each error to the lowest layer capable of resolving it before escalating to higher layers (Fig. 2). Because compile success does not guarantee runtime correctness, the feedback loop does not stop at `nvcc` success; execution outcomes are fed back into the workflow so that runtime-only faults receive targeted repair. Because WMMA failures span algorithm, API, and runtime levels, feedback is structured into three corresponding layers rather than a single undifferentiated loop. At the compile level, diagnostics from `nvcc` are fed back to GENAGENT, which re-invokes the LLM to fix syntax errors, type mismatches, and API misuse in a local self-correction loop bounded at k iterations. At the runtime level, EXECAGENT launches the compiled binary and, on failure, produces a structured diagnostic (root cause, error class, repair suggestions) that is returned to GENAGENT for code-level repair. If code-level repair does not resolve the issue, the diagnostic escalates to PLANAGENT for plan-level revision, so that deeper design errors (wrong tiling strategy, incorrect layout) are corrected at the layer where they originate. Agents communicate via typed messages routed by an asynchronous message bus with per-task tracing; a central *TaskMonitor* tracks compile/run status and signals completion. This hierarchical feedback design is the core mechanism behind TensorAgent: it closes the compile-run gap with explicit execution checks, and resolves mixed failure modes through layer-specific repair rather than conflating them in a single loop.

4.3 Planning Agent

PLANAGENT occupies the top layer of the hierarchical feedback chain: it converts an underspecified CUDA template into a structured implementation plan

that constrains downstream synthesis and reduces the algorithm-design errors identified in Sect. 3.

Domain-Specific Context and Plan Structure. The system prompt front-loads WMMA API and hardware constraints (supported fragment types, precision formats, standard tile sizes, and memory layouts) so that the plan references concrete API elements rather than generic pseudo-instructions. PLANAGENT analyzes the surrounding host code (launch dimensions, data types, stride conventions) and produces a plan of at most five actionable steps.

Feedback-Driven Revision. When plan-level escalation is triggered (Sect. 4.2), PLANAGENT receives the structured diagnostic from EXECAGENT and revises the plan accordingly. A per-task counter bounds iterations at k attempts.

4.4 Generation Agent

GENAGENT occupies the middle layer of the hierarchical feedback chain: it converts the plan into compilable CUDA and absorbs compiler-visible defects through a local feedback loop before any escalation occurs.

Constrained Code Synthesis. GENAGENT receives the plan, the original template, and any prior error feedback. Its system prompt encodes coding-level constraints (Tensor Core usage patterns, fragment loading/storing conventions, memory coalescing rules, synchronization requirements, and data-type handling) that complement the plan with code-level best practices. The LLM returns only the kernel code, delimited by triple backticks for automated extraction.

Compiler Self-feedback Loop. After splicing the generated code into the template, GENAGENT invokes `nvcc`. On failure, compiler diagnostics are appended to the prompt and the LLM is re-invoked, forming the compile-level feedback loop described in Sect. 4.2. This local loop resolves frequent compile-time faults (syntax errors, type mismatches, API misuse) without engaging the full pipeline. If the loop exhausts its k -iteration budget, the failure escalates to PLANAGENT for plan revision.

4.5 Execution Agent

EXECAGENT occupies the final layer of the hierarchical feedback chain, closing the compile-run gap identified in Sect. 3 by validating post-compile correctness and producing the structured diagnostics.

Structured Failure Diagnosis. EXECAGENT launches the compiled binary on the target GPU; exit status zero signals success under the task’s bundled checks. On a non-zero return code, EXECAGENT invokes its LLM backend with execution context (stderr, return code, CUDA and driver versions, and the original requirements) to produce a structured report: (i) failure-mode identification, (ii) root-cause attribution to a constraint layer (algorithm design, API usage, or runtime behavior), and (iii) repair suggestions.

Escalation Policy. The diagnostic is first sent to GENAGENT for code-level repair (runtime-level feedback). If regeneration fails, the failure escalates to PLANAGENT for plan revision (plan-level escalation). Shallow defects (e.g., a missing synchronization call) are thus contained at the generation layer, while deeper algorithmic problems (e.g., an incorrect tiling strategy) reach the planning layer. Escalation is not uniform across models: plan-level escalation is triggered for 66.7–98.9% of Claude Sonnet-4.6 and DeepSeek-v3.2 trials but only 2–3% of GPT-5.3-codex-high trials (Sect. 5.6, Table 6), showing that the diagnosis-and-escalation path is exercised exactly when the model cannot solve the kernel in one shot.

5 Experimental Evaluation

We assess generation with compile pass@1 and run pass@1, and compare execution time against expert baselines for successful trials. Correctness tables list three settings per model (direct prompting, TensorAgent, and CudaForge), enabling comparison against both a single-shot baseline and a state-of-the-art agentic workflow.

5.1 Benchmark Design

Task Formulation. Each task is a CUDA source file from the NVIDIA `cuda-samples` repository [19] in which the kernel body has been replaced by a placeholder. Host-side boilerplate—data allocation, memory transfers, result verification, and launch configuration—is left intact, so the LLM receives the full surrounding context and must produce only the kernel that fills the placeholder. This formulation isolates kernel logic while providing type signatures, matrix dimensions, and memory layouts that constrain a valid solution. By design, it tests the warp-level WMMA logic where the compile–run gap is most acute, but it does not exercise host integration or end-to-end performance tuning.

We assemble 15 WMMA kernel tasks from these templates, covering BF16, FP16, FP64, INT8, TF32, and mixed-precision operations in two complexity layers. The simple layer (S1–S6) contains six WMMA GEMM kernels that use core WMMA APIs. The complex layer (C1–C9) contains nine kernels that add asynchronous memory copies, cooperative-group synchronization, and optimized compute patterns (Table 1).

5.2 Experimental Setup

Models. We evaluate three LLMs: Claude Sonnet-4.6 (Anthropic), DeepSeek-v3.2 (DeepSeek), and GPT-5.3-codex-high (OpenAI). All models are accessed via API with identical prompts and temperature settings (0.1 for planning, 0.5 for generation, 0.2 for execution analysis).

Table 1. Kernel code mapping used throughout the paper: **left**, simple layer S1–S6; **right**, complex layer C1–C9; numeric order within each layer.

Simple (S1–S6)		Complex (C1–C9)	
K.	Source program	K.	Source program
S1	wmma_gemm_a_col_major_b_col_major	C1	cudaTensorCoreGemm_compute_gemm
S2	cudaTensorCoreGemm_simple_wmma_gemm	C2	dmmaTensorCoreGemm_compute_dgemm
S3	dmmaTensorCoreGemm_simple_wmma_gemm	C3	dmmaTensorCoreGemm_compute_dgemm_async_copy
S4	immaTensorCoreGemm_simple_wmma_gemm_imma	C4	dmmaTensorCoreGemm_compute_dgemm_cg_async_copy
S5	bf16TensorCoreGemm_simple_wmma_bf16gemm	C5	immaTensorCoreGemm_compute_gemm_imma
S6	tf32TensorCoreGemm_simple_wmma_tf32gemm	C6	bf16TensorCoreGemm_compute_bf16gemm
		C7	bf16TensorCoreGemm_compute_bf16gemm_async_copy
		C8	tf32TensorCoreGemm_compute_tf32gemm
		C9	tf32TensorCoreGemm_compute_tf32gemm_async_copy

Baselines. The baseline uses single-pass direct prompting; each model receives the template without planning or feedback, using the same prompt structure as GENAGENT but without the plan, compiler feedback, or execution verification. We include CudaForge [11] as a workflow-level reference, reporting TensorAgent alongside Direct and CudaForge for comparison against both a single-shot baseline and a state-of-the-art agentic workflow.

Evaluation Protocol. Following standard pass@ k evaluation [2, 7–9, 20, 21], we report compile pass@1 (compiles without errors) and run pass@1 (compiles and executes correctly) over 100 independent trials per kernel per model per method on NVIDIA A100 GPUs (CUDA 12.3).

Performance Protocol. For trials that pass runtime checks, we measure kernel execution time and compare against the expert reference kernel from the CUDA samples [19], compiled with the same matrix dimensions and launch configuration. Speedup is the expert execution time divided by the generated-kernel time; ratios above one indicate faster generated code.

5.3 Correctness Comparison

Overview. We foreground run pass@1 as the stricter final objective because it requires compilation and correct execution; on this metric TensorAgent achieves higher pass@1 than CudaForge for Claude Sonnet-4.6 (43.2% vs. 19.5% on the simple layer; 12.6% vs. 0.4% on the complex layer) and for DeepSeek-v3.2 on the simple layer (56.8% vs. 28.5%), while all three workflows tie at 0.0% run pass@1 for DeepSeek-v3.2 on the complex layer (Table 4). Although CudaForge attains higher compile pass@1 in several settings, many of its generated kernels still fail runtime checks (Tables 2–3): its hardware-guided optimization loop drives builds to succeed without closing runtime-numeric faults that determine run pass@1. TensorAgent’s advantage is therefore largest for capable-but-not-saturated models (Claude Sonnet-4.6 and DeepSeek-v3.2 on simple kernels), precisely where

compilation success is an unreliable proxy for correctness. For GPT-5.3-codex-high, which already saturates, mean run pass@1 under TensorAgent is essentially the same as under CudaForge (within one to two percentage points on each layer); and when the model lacks the required WMMA fundamentals (DeepSeek-v3.2 on the complex layer), all three workflows tie at 0% because feedback cannot supply knowledge the model does not have.

Table 2. Simple kernel results (S1–S6). Compile (C) and Run (R) pass@1 (%) over 100 trials per kernel. Within each model block, the best run pass@1 per kernel (among the three workflows) is in **bold**. Ties share bold. (In all tables, Claude = Claude Sonnet-4.6, Deepseek = DeepSeek-v3.2, GPT = GPT-5.3-codex-high. Each model appears with three rows: Baseline (single-pass prompting), TensorAgent (our framework), and CudaForge (state-of-the-art agentic CUDA workflow).)

	S1		S2		S3		S4		S5		S6	
	C	R	C	R	C	R	C	R	C	R	C	R
Claude / Baseline	19	3	25	4	13	12	20	9	45	15	3	0
Claude / TensorAgent	64	46	67	47	39	38	47	31	65	42	61	55
Claude / CudaForge	52	29	100	16	98	19	98	5	99	41	97	7
Deepseek / Baseline	32	4	71	19	73	16	58	16	81	23	67	4
Deepseek / TensorAgent	100	72	97	73	98	44	99	60	100	47	100	45
Deepseek / CudaForge	90	14	99	17	98	37	98	23	97	39	91	41
GPT / Baseline	76	76	80	80	49	48	50	49	81	79	86	81
GPT / TensorAgent	98	98	100	100	99	99	98	98	100	100	99	99
GPT / CudaForge	100	100	100	100	100	100	100	100	100	100	100	100

Simple Kernels. Table 2 compares Baseline, CudaForge, and TensorAgent on six simple kernels. Relative to Baseline, TensorAgent raises average compile pass@1 by 28.8–36.4 points and run pass@1 by 30.3–43.2 points. The run gain is larger because one-shot generation often yields code that compiles yet fails at runtime; execution-stage diagnosis closes this gap. For Claude Sonnet-4.6 and DeepSeek-v3.2, the mean run pass@1 on S1–S6 under TensorAgent exceeds that under CudaForge by 23.7 and 28.3 points, respectively. For GPT-5.3-codex-high, compile and run pass@1 are both near saturation under the agentic workflows, with only a small spread between TensorAgent and CudaForge (Table 2).

Complex Kernels. Table 3 lists per-kernel results on C1–C9; Table 4 averages them per layer. For compile pass@1, TensorAgent exceeds baseline by 30.3 % and 40.1 % for Claude Sonnet-4.6 and DeepSeek-v3.2, respectively, and also improves over Direct for GPT-5.3-codex-high (Table 3). For run pass@1, DeepSeek-v3.2 averages 0% under baseline, TensorAgent, and CudaForge. Claude Sonnet-4.6 averages 0.6% (baseline), 12.6% (TensorAgent), and 0.4% (CudaForge). For

Table 3. Complex kernel results (C1–C9). Compile (C) and Run (R) pass@1 (%) over 100 trials. Within each model block, the best run pass@1 per kernel (among the three workflows) is in **bold**. Ties share bold. There is no highlight when all three run values are 0.

	C1		C2		C3		C4		C5		C6		C7		C8		C9	
	C	R	C	R	C	R	C	R	C	R	C	R	C	R	C	R	C	R
Claude / Baseline	42	1	36	0	32	1	25	1	59	1	53	0	43	0	7	1	7	0
Claude / TensorAgent	85	6	73	14	58	17	52	17	80	19	90	7	60	13	37	10	42	10
Claude / CudaForge	99	1	98	0	97	0	97	1	98	0	96	0	97	2	96	0	96	0
Deepseek / Baseline	57	0	60	0	12	0	13	0	60	0	57	0	14	0	38	0	14	0
Deepseek / TensorAgent	98	0	93	0	44	0	68	0	97	0	88	0	52	0	87	0	59	0
Deepseek / CudaForge	92	0	84	0	41	0	52	0	88	0	78	0	43	0	82	0	39	0
GPT / Baseline	95	84	70	53	61	59	62	58	48	46	85	69	84	83	84	79	82	78
GPT / TensorAgent	99	97	99	98	100	98	100	100	95	93	99	99	100	100	98	98	99	99
GPT / CudaForge	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100

GPT-5.3-codex-high, mean run pass@1 is high under both agentic workflows and differs little between TensorAgent and CudaForge; per-kernel values are in Table 3.

Per-kernel run pass@1 is plotted in Fig. 3; Tables 2 and 3 report compile and run pass@1 for each kernel.

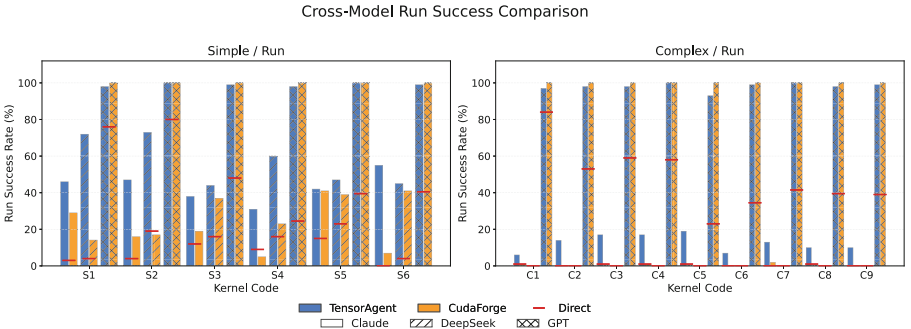


Fig. 3. Per-kernel run pass@1 (%) for each model and workflow (Direct, TensorAgent, CudaForge) on the simple and complex layers.

5.4 Performance Evaluation

Figure 4 plots per-kernel average and maximum speedup relative to expert baselines.

On simple kernels, average speedup stays close to $1.0\times$. DeepSeek-v3.2 achieves the highest single-kernel speedup on S5 (avg $1.23\times$, max $1.52\times$), followed by Claude Sonnet-4.6 on S3 (avg $1.15\times$, max $1.45\times$). Most remaining

Table 4. Summary of average compile and run pass@1 (%) across all kernels per layer. Columns are Direct prompting, TensorAgent, and CudaForge (SOTA workflow). Within each row, the best run pass@1 among the three workflows is in **bold** (no highlight when all three tie at 0%).

Layer	Model	Compile pass@1 (%)			Run pass@1 (%)		
		Baseline	TensorAgent	CudaForge	Baseline	TensorAgent	CudaForge
Simple	Claude	20.8	57.2	90.7	7.2	43.2	19.5
	Deepseek	63.7	99.0	95.5	13.7	56.8	28.5
	GPT	70.3	99.0	100.0	68.8	99.0	100.0
Complex	Claude	33.8	64.1	97.1	0.6	12.6	0.4
	Deepseek	36.1	76.2	66.6	0.0	0.0	0.0
	GPT	74.6	98.8	100.0	67.7	98.0	100.0

kernels fall within $0.87\text{--}1.06\times$, indicating that correctly generated simple kernels approach expert-level execution time.

On complex kernels, performance drops: most bars fall below $1.0\times$. Only TF32 kernels sustain competitive speedup—GPT-5.3-codex-high reaches avg $1.12\times$ on C8 (max $1.83\times$) and Claude Sonnet-4.6 avg $1.06\times$ on C9 (max $1.48\times$)—while others average $0.59\text{--}0.96\times$. Asynchronous pipelines and cooperative-group orchestration remain difficult for current generation loops to realize efficiently.

Across both layers, high run pass@1 does not by itself mean that generated kernels match expert wall-clock time. In this work we target the compile–run gap, so a runtime-correct kernel is the immediate goal, and performance tuning builds naturally on top of it. Because TensorAgent returns a runtime-correct kernel, it provides a solid starting point for such tuning: the kernel can be handed to an autotuner. A natural next step is to couple performance signals into the agent loop, so that the same hierarchical feedback also drives latency reduction on asynchronous-pipeline and cooperative-group kernels (Sect. 5.7).

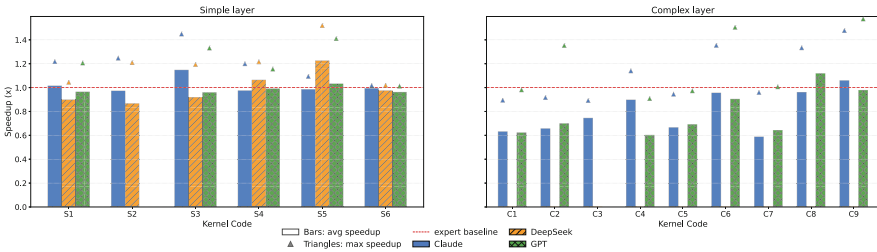


Fig. 4. Per-experiment speedup relative to the expert baseline (> 1 is faster). *Left*: simple layer; *right*: complex layer. Bar height is average speedup; triangles mark max speedup per kernel. Colors and hatches distinguish models (shared legend). The dashed red line is the $1.0\times$ expert baseline.

5.5 Ablation Study

We ablate on S1 (`wmma_gemm_a_col_major_b_col_major`, 100 trials). Table 5 compares GENAGENT alone, PLANAGENT + GENAGENT, and the full pipeline. The Gen-only row shows high compile pass@1 but very low run pass@1, consistent with the compile–run gap discussed in Sect. 3: many WMMA faults—wrong fragment layouts, missing barriers, or check failures—do not appear in `nvcc` output. Adding PLANAGENT raises run pass@1 to 40% because the structured plan commits to tile shape, WMMA steps, and layout or synchronization requirements before synthesis, so GENAGENT follows a coherent algorithm instead of one-shot guessing from the template. Including EXECAGENT brings run pass@1 to 67%: execution on the target GPU is the only reliable signal for several failure modes, and structured diagnostics let the workflow iterate on those runtime-only errors rather than stopping at a successful build.

Table 5. Ablation study on `wmma_gemm_a_col_major_b_col_major` (S1, 100 trials). Rows add agents from Gen-only to the full pipeline. Best run pass@1 in **bold**.

Configuration	Agents	Compile (%)	Run (%)
Gen only	GENAGENT	90	2
Plan + Gen	PLANAGENT + GENAGENT	98	40
Full pipeline	PLANAGENT + GENAGENT + EXECAGENT	100	67

5.6 Efficiency and Practical Cost

Table 6 reports practical-cost statistics: the mean number of LLM calls per trial, the mean number of GENAGENT (compile-repair) calls per trial, the fraction of trials that trigger plan-level escalation, and the mean end-to-end wall-clock time. Direct prompting issues exactly one LLM call per trial by construction. Iteration count scales with task difficulty and inversely with model competence, while absolute wall-clock is set by the backend. For GPT-5.3-codex-high, which usually succeeds on the first attempt, TensorAgent averages only 2.5 calls per trial and escalates the plan in 2–3% of trials, so its overhead over direct prompting is small. For Claude Sonnet-4.6 and DeepSeek-v3.2, which need substantial repair, the framework averages 9.9–13.5 calls per trial and escalates in 66.7–98.9% of trials. On the complex layer the agent loop runs to nearly the same budget for both models (13.3 vs 13.5 calls; the plan-revision cap saturates on ~90% of trials in both), yet mean wall-clock differs by 2.8× (402 s for Claude Sonnet-4.6 vs 1,143 s for DeepSeek-v3.2); the gap reflects per-call latency on the underlying API (~30 s vs ~85 s) rather than additional repair work.

Table 6. Practical cost of TensorAgent, averaged over all logged trials per layer. **Calls**: mean LLM calls per trial (Direct = 1). **Repair**: mean GENAGENT compile-repair calls per trial. **Escal.**: fraction of trials triggering plan-level escalation. **Wall**: mean end-to-end wall-clock time per trial (includes compilation, execution, and API latency).

Model	Layer	Calls	Repair	Escal. (%)	Wall (s)
Claude	Simple	9.9	7.2	79.0	173
	Complex	13.3	9.5	94.9	402
Deepseek	Simple	10.4	5.5	66.7	386
	Complex	13.5	8.3	98.9	1143
GPT	Simple	2.5	1.5	3.0	178
	Complex	2.5	1.4	2.0	279

5.7 Future Work

TensorAgent opens several promising directions for extending hierarchical-feedback agents to broader Tensor Core programming. First, we plan to move beyond the warp-level WMMA setting toward end-to-end kernel development: lifting the fixed host code, opening up free data-layout and tiling choices, and integrating host-device boundaries so that the same layered diagnosis can guide production-scale kernel workflows. Second, we will extend the framework from the compile-run gap to performance tuning by coupling latency and occupancy signals into the agent loop, so that hierarchical feedback drives not only correctness but also wall-clock speedup, naturally connecting TensorAgent with autotuners. Third, we see strong potential in combining TensorAgent with complementary repair mechanisms such as constraint-aware decoding, symbolic verification, and reinforcement-based repair, which can further sharpen each layer of the feedback hierarchy. Finally, our cross-model study suggests a model-adaptive variant: calibrating the granularity and aggressiveness of feedback to the underlying LLM’s capability, so that capable-but-not-saturated models, saturated models, and models lacking WMMA fundamentals can each be served by an appropriately tuned agent loop.

6 Conclusion

We presented TensorAgent, a multi-agent framework for WMMA Tensor Core code generation that aligns planning, synthesis, and execution diagnosis with the layered failure structure of WMMA programming and links them through hierarchical feedback that repairs failures at the layer where they originate. Across 15 benchmarks, six data-type settings, two complexity tiers, three LLMs, and 9,000 trials, TensorAgent improves compile and run pass@1 over direct prompting and outperforms the agentic CudaForge [11] baseline on run pass@1 for capable-but-not-saturated models, while remaining inexpensive when the model

already succeeds (Sect. 5.6). As discussed in Sect. 5.7, coupling performance signals into the agent loop and extending beyond WMMA-centric templates are the main directions ahead, broadening hierarchical feedback from correctness to end-to-end Tensor Core performance.

Acknowledgment. We thank the anonymous reviewers for their constructive and insightful feedback. This work was supported in part by the National Natural Science Foundation of China (NSFC) under grant 62472462 and fund GBA NCTI-ISDC[2026]048.

AI Use Statement. The technical solution and experimental design presented in this paper were independently completed by the authors. AI tools were used solely for language polishing and formatting optimization, and did not participate in the development of the research ideas or core content.

References

1. NVIDIA Corporation: CUDA C++ programming guide: Warp matrix functions (2024). <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>
2. Chen, M., Tworek, J., Jun, H., et al.: Evaluating large language models trained on code (2021). <https://arxiv.org/abs/2107.03374>
3. Li, R., et al.: StarCoder: may the source be with you! (2023). <https://arxiv.org/abs/2305.06161>
4. Rozière, B., et al.: Code Llama: open foundation models for code (2024). <https://arxiv.org/abs/2308.12950>
5. Ouyang, A., et al.: KernelBench: can LLMs write efficient GPU kernels? (2025). <https://arxiv.org/abs/2502.10517>
6. Zhu, J., et al.: CUDABench: benchmarking LLMs for Text-to-CUDA generation (2026). <https://arxiv.org/abs/2603.02236>
7. Huang, D., Zhang, J.M., Luck, M., Bu, Q., Qing, Y., Cui, H.: AgentCoder: multi-agent-based code generation with iterative testing and optimisation (2024). <https://arxiv.org/abs/2312.13010>
8. Islam, M.A., Ali, M.E., Parvez, M.R.: MapCoder: multi-agent code generation for competitive problem solving. In: Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). Association for Computational Linguistics, August 2024. <https://aclanthology.org/2024.acl-long.269/>
9. Zhang, K., Li, J., Li, G., Shi, X., Jin, Z.: CodeAgent: enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. In: Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). Association for Computational Linguistics, August 2024. <https://aclanthology.org/2024.acl-long.737/>
10. Islam, M.A., Ali, M.E., Parvez, M.R.: CodeSim: multi-agent code generation and problem solving through simulation-driven planning and debugging. In: Findings of the Association for Computational Linguistics: NAACL 2025. Association for Computational Linguistics, April 2025. <https://aclanthology.org/2025.findings-naacl.285/>

11. Zhang, Z., Wang, R., Li, S., Luo, Y., Hong, M., Ding, C.: CudaForge: an agent framework with hardware feedback for CUDA kernel optimization (2025). <https://arxiv.org/abs/2511.01884>
12. Wei, J., et al.: Chain-of-thought prompting elicits reasoning in large language models (2023). <https://arxiv.org/abs/2201.11903>
13. Chen, X., Lin, M., Schärli, N., Zhou, D.: Teaching large language models to self-debug (2023). <https://arxiv.org/abs/2304.05128>
14. Ridnik, T., Kredo, D., Friedman, I.: Code generation with AlphaCodium: from prompt engineering to flow engineering (2024). <https://arxiv.org/abs/2401.08500>
15. Dai, W., et al.: CUDA agent: large-scale agentic RL for high-performance CUDA kernel generation (2026). <https://arxiv.org/abs/2602.24286>
16. Guo, T., et al.: Large language model based multi-agents: a survey of progress and challenges (2024). <https://arxiv.org/abs/2402.01680>
17. Tran, K.T., Dao, D., Nguyen, M.D., Pham, Q.V., O’Sullivan, B., Nguyen, H.D.: Multi-agent collaboration mechanisms: a survey of LLMs (2025). <https://arxiv.org/abs/2501.06322>
18. Adapala, S.T.R.: Cognitive load limits in large language models: benchmarking multi-hop reasoning (2025). <https://arxiv.org/abs/2509.19517>
19. NVIDIA Corporation: CUDA samples (2024). <https://github.com/NVIDIA/cuda-samples>
20. Austin, J., et al.: Program synthesis with large language models (2021). <https://arxiv.org/abs/2108.07732>
21. Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., et al.: Competition-level code generation with AlphaCode. *Science* **378**(6624), 1092–1097 (2022). <https://doi.org/10.1126/science.abq1158>