

AscendKernelGen: LLM-Driven Kernel Generation for NPUs

Xinzi Cao^{1,3*}, Jianyang Zhai^{1,3*}, Pengfei Li^{2*}, Zhiheng Hu^{2*},
Cen Yan², Bingxu Mu², Guanghuan Fang², Bin She², Jiayu Li², Yihan Su²,
Dongyang Tao², Feidiao Yang¹, Chang-Dong Wang³, Yutong Lu³,
Weicheng Xue^{1†}, Bin Zhou^{1†}, Yonghong Tian^{1,4†}

¹Pengcheng Laboratory, ²Huawei, ³Sun Yat-sen University, ⁴Peking University

Correspondence: xuewch@pcl.ac.cn, senosy@gmail.com, yhtian@pku.edu.cn

Abstract

Neural Processing Units (NPUs) are critical for AI infrastructure, yet developing kernels remains a bottleneck due to the complexity of vendor-specific Domain-Specific Languages (DSLs). While LLMs excel in general coding, they fail to meet the stringent constraints of NPU development, showing a near-zero success rate on complex kernels in our preliminary study. To address these challenges, we present AscendKernelGen, the first comprehensive framework for NPU kernel development, marking a pioneering effort in this field. This framework consists of three interconnected components: (1) Ascend-CoT, the first dataset in the NPU kernel domain that incorporates chain-of-thought reasoning from real-world kernel implementations; (2) KernelGen-LM, a domain-adaptive model trained on this novel dataset using supervised fine-tuning and reinforcement learning; and (3) NPUIKernelBench, the first benchmark platform designed to evaluate the compilation, correctness, and performance of generated NPU kernels. Experimental results demonstrate that our approach dramatically bridges the gap in hardware-specific coding: compilation success on complex Level-2 kernels improves from 0% to 95.5% (Pass@10), with 64% functional correctness. AscendKernelGen is available at [AscendKernelGen](#) and [NPUIKernelBench](#).

1 Introduction

The advancement of Artificial Intelligence (AI), particularly deep learning, has fundamentally reshaped modern computing architectures. To meet the ever-increasing demand for computational efficiency and throughput, domain-specific accelerators have emerged as a key solution. In this setting, Neural Processing Units (NPUs), such as Ascend NPU (Liao et al., 2021; Xue et al., 2024;

Wróblewski et al., 2025), have become an important component of contemporary AI infrastructure. However, the practical efficiency of these accelerators is determined not only by hardware capabilities, but also by the quality of the underlying compute kernels. Therefore, developing high-performance, hardware-adapted kernels is a prerequisite for unlocking the potential of NPU platforms.

Writing such kernels remains extremely challenging. NPU kernel development typically relies on highly specialized, vendor-specific domain-specific languages (DSLs), such as AscendC, which requires deep expertise in hardware architecture. This includes fine-grained management of hierarchical memory systems (e.g., global versus on-chip memory), carefully designed data tiling strategies, asynchronous pipeline programming for overlapping computation and data movement, and explicit utilization of vector (SIMD) and matrix (Cube) execution units. The resulting learning curve is steep, making manual kernel development time-consuming, costly, and error-prone. Consequently, kernel optimization has become a major bottleneck in the rapid iteration and deployment of AI applications on NPU architectures.

Although recent efforts have explored LLM-based kernel generation, existing approaches suffer from two fundamental limitations. First, general-purpose models lack the domain knowledge required to satisfy hardware-specific constraints. As shown by our zero-shot evaluation (Table 1), even state-of-the-art LLMs frequently hallucinate non-existent APIs (e.g., `AscendC::Softmax`) or misuse core interfaces, leading to widespread compilation failures and near-zero execution success on complex L2/L3 kernels. Second, generic code corpora are ineffective for this task: NPU kernels rely on specialized DSLs with strict syntax and architecture-dependent optimization patterns, while high-quality NPU-specific training data remains scarce. Addressing these challenges

*These authors contributed equally.

†Corresponding authors.

Table 1: Zero-shot performance of LLMs on NPU kernel generation tasks.

Model	Task Type	Compilation Success Rate	Functional Correctness (Post-Compilation)
Qwen3-8B	L1 (Simple)	8.22%	1.08%
	L2/L3 (Complex)	1.39%	0.00%
Qwen2.5-Coder-7B	L1 (Simple)	9.19%	0.47%
	L2/L3 (Complex)	0.40%	0.00%
Llama3.1-8B	L1 (Simple)	23.97%	0.69%
	L2/L3 (Complex)	19.44%	0.00%
Mistral-7B	L1 (Simple)	0.00%	0.00%
	L2/L3 (Complex)	0.00%	0.00%

therefore requires domain-adaptive post-training on reasoning-rich, hardware-aware datasets rather than improved prompting alone.

To address these challenges, we propose **AscendKernelGen**, a generation–evaluation framework for low-level NPU kernel development. The core idea of AscendKernelGen is to bridge the gap between general-purpose LLMs and hardware-specific kernel programming through domain-adaptive data, training, and evaluation. Specifically, we introduce a reasoning-oriented dataset to expose models to kernel-level design logic, adopt domain-adaptive post-training to improve execution robustness, and establish a rigorous benchmark to systematically assess compilation, correctness, and performance. Together, these components enable reliable and scalable evaluation of LLM-based NPU kernel generation beyond surface-level code validity. Our main contributions are threefold:

1. A reasoning-oriented dataset for NPU kernel generation. We construct *Ascend-CoT*, a dataset curated from real-world AscendC kernels that captures structured reasoning patterns such as pipeline construction, synchronization, and arithmetic logic.
2. Domain-adaptive post-training for kernel synthesis. We propose a post-training strategy tailored to NPU kernel generation, producing *KernelGen-LM* with significantly improved compilation success and functional correctness over general-purpose code models.
3. A comprehensive benchmark for NPU kernel evaluation. We introduce *NPUKernelBench*, which evaluates generated kernels in terms of compilation, correctness, and performance under both static- and dynamic-shape settings.

2 Related Works

LLM-based Code Generation. Large Language Models have shown strong performance

in general-purpose code generation across mainstream languages, as demonstrated by systems such as Codex (Yadav and Mondal, 2025), AlphaCode (Novikov et al., 2025), and Qwen2.5-Coder (Hui et al., 2024). However, their effectiveness degrades substantially on domain-specific tasks (Gu et al., 2025), particularly hardware programming that requires explicit memory management and architecture-dependent semantics.

Recent work has explored LLM-based hardware kernel generation, primarily for mature ecosystems like CUDA and TPUs. Approaches range from prompting-based methods (Chen et al., 2025; Lange et al., 2025) and supervised fine-tuning (Fisches et al., 2025), to reinforcement learning with execution feedback (Baronio et al., 2025; Li et al., 2025; Woo et al., 2025) and agentic frameworks (Guo et al., 2025; Dong et al., 2025; Zhang et al., 2025). Despite these advances, existing work predominantly targets well-documented platforms with abundant training data. LLM-driven kernel generation for emerging NPU platforms with domain-specific programming models remains largely underexplored, motivating our work on domain-adaptive training strategies and comprehensive evaluation frameworks.

Traditional Kernel Optimization Approaches.

Prior to LLMs, kernel optimization relied primarily on compiler-based and auto-tuning methods. Auto-tuning frameworks like TVM (Chen et al., 2018) and Ansor (Zheng et al., 2020) search over optimization spaces for diverse hardware backends, while domain-specific languages such as Halide (Ragan-Kelley et al., 2013) and Triton (Tillet et al., 2019) decouple algorithm specification from execution scheduling. Compiler pipelines like TensorFlow XLA (Snider and Liang, 2023) and PyTorch 2.0 (Ansel et al., 2024) transform computation graphs into optimized kernels. More specialized approaches leverage polyhedral compilation (Baghdadi et al., 2019; Zhao et al., 2021) or platform-specific libraries (Markidis et al., 2018) for particular hardware workload combinations. While achieving high performance, these methods typically require substantial expert knowledge, incur high development costs, and offer limited cross-architecture generalization, motivating more flexible learning-based approaches.

Evaluation Benchmarks for Kernel Generation.

Standard benchmarks such as HumanEval (Yadav and Mondal, 2025) and MBPP (Austin et al., 2021)

primarily target functional correctness, making them insufficient for hardware kernel generation, where compilation feasibility and performance are equally important. Hardware-oriented benchmarks, including KernelBench (Ouyang et al., 2025) and TritonGym (Guan et al., 2026), extend evaluation to compilation and performance. However, most existing benchmarks require full host-device code generation for all tasks, limiting flexibility across different kernel abstraction levels. Moreover, comprehensive evaluation frameworks tailored to NPU kernel generation remain lacking.

3 Challenges in NPU Kernel Generation

We identify challenges in programming abstractions of low-level NPU kernels, complicating LLM-based code generation. These challenges stem from key aspects of NPU kernel design. A low-level NPU kernel is a statically structured, data-parallel program that encodes global data partitioning, asynchronous pipeline stages, and synchronization semantics. As a result, each kernel instance operates on a distinct data slice defined by block indices, necessitating explicit computation of memory offsets and boundary conditions. The computation is organized as an asynchronous pipeline, mapping data movement, arithmetic operations, and write-back to independent hardware units. Additionally, the kernel encodes producer-consumer relationships through synchronization primitives, establishing an execution schedule that couples arithmetic, data movement, and control flow. Consequently, this abstraction imposes several reasoning requirements:

Global semantic consistency. Kernel correctness frequently depends on auxiliary parameters such as block indices, tiling factors, and boundary sizes, which are computed externally and passed in at runtime. These parameters are referenced across multiple pipeline stages and jointly govern memory access and synchronization behavior. Generating correct kernel code therefore requires maintaining semantic consistency across distant code regions, demanding reasoning over long-range dependencies rather than local pattern completion.

Explicit synchronization reasoning. Asynchronous pipeline stages communicate through manually inserted synchronization primitives. Correct execution relies on precise pairing and ordering of these operations to enforce producer-consumer relationships, and even minor

misplacement can introduce deadlocks or silent data hazards. This makes correctness dependent on reasoning about dynamic execution ordering rather than simple linear instruction sequences.

Boundary-sensitive arithmetic reasoning. Because tensor dimensions are often not perfectly aligned with hardware-preferred block sizes, kernel code must explicitly handle boundary cases through offset computation, loop-bound reasoning, and masking logic. This requires precise arithmetic reasoning over indices, validity conditions, and partial-tile execution. Even small mistakes in these calculations can lead to incorrect outputs or invalid memory accesses, making boundary handling a fundamental challenge in kernel generation.

Layout-aware representation reasoning. Kernel execution often involves physical data layouts that are specialized for particular execution units or memory movement patterns. As a result, generated code must correctly manage layout transformations and address computations while preserving the logical semantics of the original tensors. This separation between abstract tensor meaning and hardware-oriented representation requires reasoning beyond shape-level manipulation and remains difficult for general-purpose code models to infer implicitly.

4 AscendKernelGen

The framework integrates three key components: the *Ascend-CoT* dataset for hardware-grounded reasoning (§4.1), the *KernelGen-LM* model for kernel generation (§4.2), and the *NPUKernelBench* benchmark for execution-based evaluation (§4.3). A system overview is shown in Figure 1.

4.1 Ascend-CoT Dataset

To equip LLMs with reasoning capabilities for NPU kernel generation, we construct a multi-source chain-of-thought dataset that integrates AscendC domain knowledge with general reasoning supervision. The dataset exposes models to kernel structures, tiling strategies, memory movement, API constraints, and correctness reasoning in hardware-specific contexts.

To equip LLMs with reasoning capabilities for NPU kernel generation, we construct *Ascend-CoT*, a multi-source chain-of-thought dataset that integrates AscendC domain knowledge with general reasoning supervision. The dataset exposes models to kernel structures, tiling strategies, memory

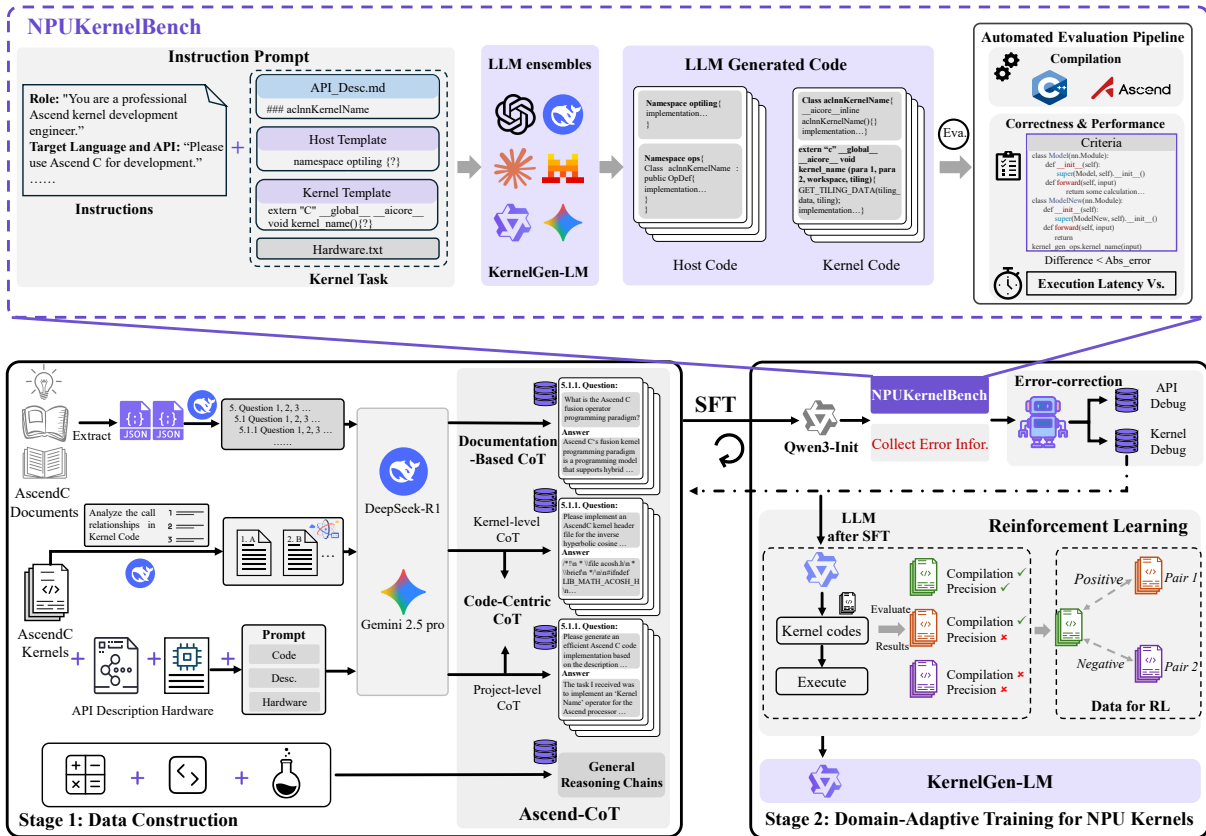


Figure 1: System overview of AscendKernelGen, depicting the data construction, LLM training, and hardware-grounded evaluation pipeline for NPU kernel generation.

movement, API constraints, and correctness reasoning in hardware-specific contexts. As illustrated in Figure 1, *Ascend-CoT* comprises three components: documentation-based chain-of-thought (CoT) distilled from AscendC manuals, kernel-centric CoT derived from real-world kernels, and general CoT included to improve transfer and generalization. In total, the raw corpus contains 83,916 samples and is publicly available at [Ascend-CoT](#).

The raw dataset exhibits a long-tailed length distribution. Approximately 99.1% of samples have input sequences shorter than 11.1k tokens, although the maximum input length reaches about 111k tokens. On the output side, 85.1% of samples are within 21.9k tokens and an additional 6.3% fall between 21.9k and 43.8k tokens, while the maximum output length reaches about 219k tokens. Before supervised fine-tuning, we apply length-aware preprocessing: samples exceeding 32k tokens are discarded, and short samples are packed into fixed-length training sequences using boundary-aware isolation to prevent interference across concatenated examples. After preprocessing, the final SFT corpus consists of 9,955 sequences of length 32k.

4.1.1 Documentation-Based CoT Supervision

To address the complexity of AscendC programming abstractions and strict API constraints, we construct documentation-based CoT supervision from authoritative AscendC manuals. These sources cover the operator programming model, API semantics, memory hierarchy, and performance guidelines, providing a principled foundation for low-level kernel reasoning.

Rather than treating documentation as plain text, we adopt a knowledge-instruct paradigm (Ovadia et al., 2025) that transforms manual content into question-answer pairs with explicit reasoning traces. This supervision encourages models to reason about API usage, memory rules, and architectural constraints, mitigating hallucinations in low-level code generation.

4.1.2 Code-Centric CoT Supervision

Correct kernel generation requires understanding the interaction between host-side tiling logic and device-side kernel execution. To capture this, we construct Code-centric CoT data from real-world AscendC operator implementations, which consist of two key components: *kernel-level CoT* and

project-level CoT.

Kernel-level CoT focuses on kernel-internal reasoning. Given a standalone kernel file, we generate targeted question–answer pairs with explicit reasoning traces that explain key design choices, including function usage, memory management, tiling assumptions, and pipeline organization. These reasoning traces are intended to expose the intermediate logic that connects low-level implementation details with the underlying execution semantics, helping the model learn not only what the kernel does, but also why specific implementation decisions are required.

In contrast, *project-level CoT* targets industrial-grade operators that interleave host-side tiling logic with device-side kernel invocation. To enable focused analysis, we first decompose each operator into logically pure host–kernel pairs, each corresponding to a single execution scenario with fixed shape, tiling parameters, and data type. For each pair, we prompt the model with the kernel code, host-side logic, and relevant API documentation to generate structured reasoning that explains tiling decisions and ensures consistency between host metadata and kernel execution behavior. This supervision exposes cross-boundary dependencies that are difficult to infer from isolated kernel code.

4.1.3 General CoT Supervision

To ensure effective training while maintaining general reasoning ability, we augment our kernel dataset with high-quality open-source chain-of-thought corpora in mathematics, code reasoning, and scientific problem solving (Liu et al., 2025; voidful, 2025). A unified filtering and normalization pipeline is employed to achieve consistency across diverse sources, utilizing language normalization, length-based filtering, and perplexity-based quality control. This approach enables the model to maintain general problem-solving skills while meeting the specialized demands of NPU kernel generation.

4.2 KernelGen-LM: Domain-Adaptive Training for NPU Kernels

To enable reliable NPU kernel generation under strict programming constraints, we introduce *KernelGen-LM*, a publicly available domain-adapted model designed to internalize AscendC-specific API semantics and complex execution dependencies. Our approach employs a two-stage optimization strategy combining Supervised Fine-

Tuning (SFT) and Reinforcement Learning (RL). The SFT stage injects structured knowledge and error-derived correction signals to enforce fundamental compilation and numerical correctness, while the RL stage utilizes execution-based feedback to distinguish high-performance implementations among candidates.

4.2.1 Error-Derived Supervised Fine-Tuning

Although standard SFT establishes basic structural competence, models trained solely on static data remain prone to compilation failures and silent numerical errors. To improve robustness prior to reinforcement learning, we introduce an *error-derived supervision* mechanism that augments SFT with correction signals mined from real execution failures, shown in Figure 2. This supervision targets two complementary failure modes in LLM-generated kernels: (i) API-level misuse that obstructs compilation, and (ii) kernel-level numerical inconsistencies manifesting during execution.

API-level error correction. To resolve compilation failures caused by API misuse, we construct correction samples using error logs clustered by signature and augmented with retrieved documentation. This supervision trains the model to diagnose failure causes and synthesize valid corrections given specific compiler feedback, effectively mitigating semantic hallucinations where syntactically correct calls violate usage constraints.

Kernel-level error correction. To address silent numerical inconsistencies (e.g., memory staging or tiling errors) in executable kernels, we employ ground-truth-guided reconstruction. By pairing failing kernels with verified reference implementations, we generate reconstruction-oriented CoT supervision. This guides the model to analyze reasoning flaws and regenerate correct execution logic.

4.2.2 Reinforcement Learning with Execution-Based Preferences

After error-derived supervised fine-tuning, the model is able to generate kernels that largely satisfy compilation constraints and basic numerical correctness. Nevertheless, for a given specification, multiple executable implementations may still exist, differing in subtle execution properties such as memory access patterns, accumulation order, or numerical stability. These distinctions are difficult to capture through deterministic supervision alone.

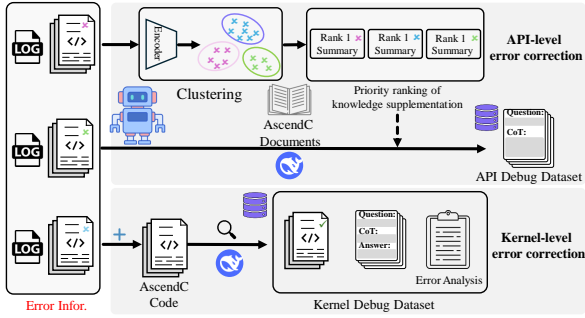


Figure 2: Error-Derived Supervision for API- and Kernel-Level Error Correction.

To further distinguish high-quality kernels from merely syntactically valid ones, we introduce a reinforcement learning stage grounded in execution feedback, as illustrated in the bottom-right panel of Figure 1. For each task, the SFT-tuned model samples multiple candidates, which are validated on hardware and categorized into three levels: fully correct (Green), executable but numerically incorrect (Orange), and compilation failures (Purple). These outcomes are then constructed into a preference dataset of pairwise comparisons (e.g., Green \succ Orange/Purple). These pairs define a relative preference signal indicating which generations are more desirable. The resulting preferences are then used to update the model, encouraging behaviors that consistently lead to more stable and accurate kernel implementations.

This RL stage benefits from the preceding error-derived SFT process. By eliminating a large fraction of invalid or numerically incorrect candidates beforehand, supervised correction significantly narrows the policy search space and stabilizes preference learning. As a result, reinforcement learning focuses on fine-grained optimization among valid kernels rather than recovering from catastrophic execution failures, leading to improved training efficiency and convergence stability.

4.3 NPUKernelBench: Evaluation of NPU Kernel Generation

We introduce [NPUKernelBench](#), an end-to-end benchmark for evaluating LLMs on NPU kernel generation under real hardware constraints. Given a kernel task specification, the model is required to generate executable host-side and kernel-side code, which is subsequently compiled and executed on actual NPU hardware. This design not only grounds evaluation in full compilation and execution but also enables reproducible and hardware-faithful

assessments of kernel generation. In order to disentangle sources of difficulty, kernel tasks are categorized into three algorithmic complexity levels (L1~L3). Representative kernels and task statistics for each level are summarized in Table 5 of Appendix A.

Moreover, NPUKernelBench distinguishes between static-shape and dynamic-shape evaluation settings. Specifically, static-shape tasks assess specialization and optimization capabilities for fixed tensor dimensions, whereas dynamic-shape tasks evaluate robustness under runtime-varying shapes, necessitating correct host-side shape inference and tiling logic. Consequently, these settings effectively characterize both specialization and generalization behaviors of LLMs in kernel generation.

NPUKernelBench is intentionally designed as a controlled evaluation environment rather than an unconstrained code-generation benchmark. By standardizing task specifications, code templates, and verification protocols, it reduces ambiguity in requirement interpretation and interface design, allowing the evaluation to focus on whether the model has internalized the core challenges of NPU programming, including asynchronous execution, explicit synchronization, boundary handling, and hierarchical memory management. This design makes the benchmark particularly suitable for measuring the transition from failure to correct and efficient kernel generation under strict external verification, while remaining complementary to broader benchmarks that emphasize integration complexity and environmental variability.

5 Experiment

5.1 Experimental Setups

LLMs. We employ specific large language models tailored to distinct pipeline stages, primarily DeepSeek-R1 ([DeepSeek-AI, 2025](#)) and Gemini 2.5 Pro ([Gemini-Team, 2025](#)). Please refer to Table 6 in Appendix B for a detailed summary of their functional roles and specific configurations.

Training setting. We implement a two-stage training strategy encompassing SFT and RL. For the SFT phase, Qwen3-32B ([Team, 2025](#)) serves as the base model. Additionally, to verify scalability and versatility, we conduct parallel SFT experiments on the wider Qwen family (1.7B–32B, non-MoE architecture) and the code-specific Qwen3-Coder-30B (MoE architecture). All SFT runs adhere to a unified configuration, benchmarking both

full fine-tuning and LoRA strategies. The second stage employs Direct Preference Optimization (DPO) (Rafailov et al., 2023) for alignment. Please refer to Table 6 in Appendix B for the exhaustive list of hyperparameters and experimental settings.

Repeated runs and result reporting. Each configuration was evaluated at least three times on real NPU hardware under identical settings, and the reported results are the mean values across runs. Compilation and correctness outcomes were consistent across repeated trials, while runtime-related metrics exhibited only minor variation, confirming the stability and reproducibility of the results.

Evaluation Metrics. Each kernel includes a Python reference, which serves as the ground truth for our three evaluation metrics. We first measure *Compilation Rate (CR)* and *Execution Rate (ER)* to evaluate code validity. These are quantified using $\text{pass}@k$, defined as the success rate given k generation attempts per task. Beyond correctness, we report *Speedup* to assess efficiency, calculated by normalizing the latency of generated kernels against vendor-optimized baselines.

5.2 Main Results

We evaluate kernel generation performance across training stages: base models, supervised fine-tuning, and reinforcement learning. Performance consistently improves with training, showing enhanced robustness on more challenging kernels. Quantitative results for various sampling budgets are summarized in Table 2 and Figure 3.

5.2.1 Compilation and Execution

Table 2 highlights a significant performance bottleneck tied to kernel complexity. While the base Qwen3-32B model achieves moderate success on Level 1 tasks, it fails entirely on Level 2 and 3 kernels (0.0% ER), where even successful compilation (e.g., 50% Pass@100 CR for Level 3) does not guarantee correctness. SFT effectively bridges this gap by instilling structural knowledge, boosting mean Pass@100 CR to nearly 100% and significantly recovering ER for Level 1 and 2 tasks. RL further refines these candidates, raising the mean Pass@100 ER to 88.89% and achieving the highest speedups ($1.98\times$ for Level 2). This progression confirms that while SFT ensures syntactic validity, the SFT+RL hierarchy is essential for mastering complex functional logic in NPU kernel generation.

Figure 3 shows Pass@1 accuracy on representative kernels for Qwen-32B under three training

stages. Compared to the base model, SFT yields a substantial improvement, raising the average Pass@1 from 7.92% to 26.26%, which indicates that supervised alignment effectively equips the model with basic kernel synthesis capabilities. Applying RL on top of SFT further improves accuracy to 33.46%, with more pronounced gains on complex kernels. This consistent progression confirms that RL complements SFT by refining fine-grained execution logic beyond syntactic correctness.

5.2.2 Speedup

Beyond correctness, Table 2 demonstrates the runtime efficiency of the generated kernels. While the base model fails to match expert baselines, yielding only $0.60\times$ speedup on Level 1 and failing completely on higher levels, the proposed training pipeline delivers substantial gains. SFT not only recovers execution but also optimizes latency, achieving a $1.50\times$ speedup on Level 2 tasks, effectively surpassing expert implementations. RL further refines these kernels, pushing Level 2 speedup to $1.86\times$ while maintaining comparable performance on Level 1 ($0.61\times$). This progression indicates that the model learns to generate not just syntactically valid code, but also efficient parallelization patterns, particularly on Level 2 tasks where it exceeds expert-written baselines.

5.3 Ablation Analysis of SFT

5.3.1 Model Scale Sensitivity

As shown in Figure 4, increasing model scale leads to clear performance gains on Level 1 and Level 2 tasks, indicating that larger models better capture standard kernel patterns. In contrast, Level 3 tasks remain highly challenging, with most models failing to achieve non-trivial execution rates. However, 32B is the only scale that demonstrates measurable compilation success on Level 3, suggesting that sufficient parameter scale is a prerequisite for reasoning over the complex control flow and memory dependencies of complicated NPU kernels.

5.3.2 Comparison of Fine-tuning Strategies

Table 3 compares full fine-tuning (FFT) and LoRA (Hu et al., 2022) on an 8B model. FFT consistently outperforms LoRA, with mean compilation and execution rates improving from 40.29%/13.55% to 55.32%/22.13%. This gap is attributed to the complexity of NPU kernel generation, which requires precise modeling of hardware constraints and execution logic. The limited capac-

Table 2: Evaluation results of kernel generation on NPUKernelBench across different sampling budgets (k). We report Compilation Rate (CR), Execution Rate (ER), and overall Speedup for generated kernels.

Model	Level	Pass@1		Pass@10		Pass@100		Speedup (\times)
		CR (%)	ER (%)	CR (%)	ER (%)	CR (%)	ER (%)	
Qwen3-32B	Level 1	38.08	17.39	71.62	48.84	75.00	66.67	0.60
	Level 2	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Level 3	1.83	0.0	15.43	0.0	50.00	0.0	0.0
	Mean	25.59	11.59	49.46	32.56	55.56	44.44	0.60
Qwen3-32B + SFT	Level 1	84.83	38.28	99.97	86.82	100	94.44	0.56
	Level 2	60.5	8	96.54	40.48	100	75	1.50
	Level 3	17	0.17	63.75	1.67	100	16.67	0.00
	Mean	71.89	27.31	95.18	67.06	100	81.48	0.75
Qwen3-32B + SFT + RL	Level 1	82.17	43.28	99.92	93.89	100	100	0.61
	Level 2	62.75	14.25	95.49	64.28	100	91.67	1.86
	Level 3	14.67	0.17	61.12	1.67	100	16.67	0.00
	Mean	70.35	32.04	94.62	77.06	100	88.89	0.87

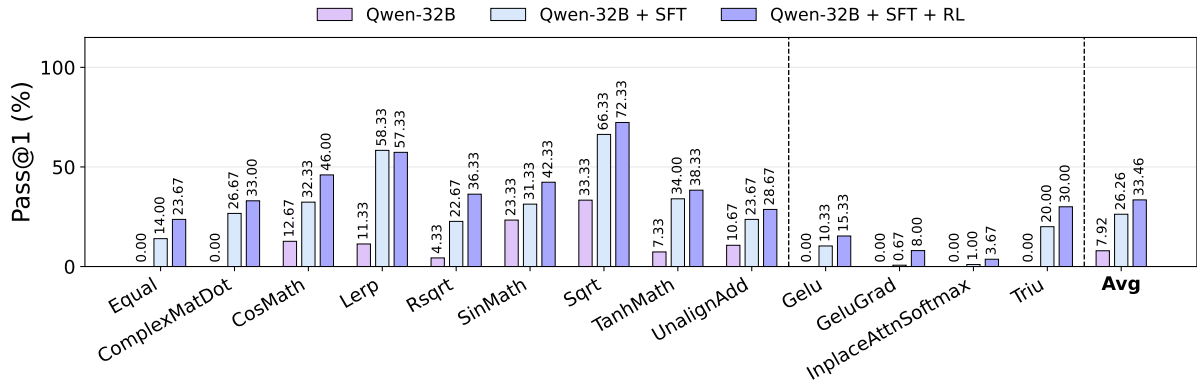


Figure 3: Pass@1 results for representative kernels across training stages.

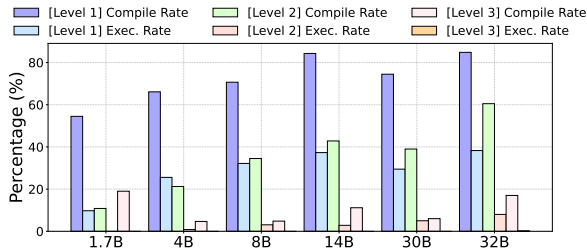


Figure 4: Scaling behavior of KernelGen-LM on NPU kernel generation tasks with increasing difficulty.

ity of low-rank updates proves insufficient, while full fine-tuning allows for comprehensive parameter adjustments, resulting in higher correctness and better runtime performance ($0.95\times$ vs. $0.48\times$).

5.3.3 Training Data Composition Analysis

Figure 5a shows that kernel-centric CoT is crucial for kernel generation; its removal leads to a significant drop in compilation success (e.g., Level1 below 20%), highlighting the need for domain-

Table 3: Comparison of full fine-tuning and LoRA fine-tuning on Qwen3-8B.

Model	Level	Compile Rate (%)	Exec. Rate (%)	Speedup (\times)
LoRA-tuning	Level 1	54.94	20.10	0.48
	Level 2	14.58	0.67	0.52
	Level 3	3.83	0.00	-
	Mean	40.29	13.55	0.48
Full-tuning	Level 1	70.67	32.17	0.58
	Level 2	34.50	3.08	2.77
	Level 3	4.83	0.00	-
	Mean	55.32	22.13	0.95

specific syntax. General CoT also supports performance, as its absence consistently degrades compilation rates, indicating that general code reasoning complements kernel synthesis. In contrast, excluding Documents has minimal impact, suggesting the model relies more on code structure than textual descriptions. Additionally, Figure 5b shows that performance improves with the training data ratio, with rapid gains at low scales (0%–25%) followed by steady growth, emphasizing the importance of large-scale, high-quality training data.

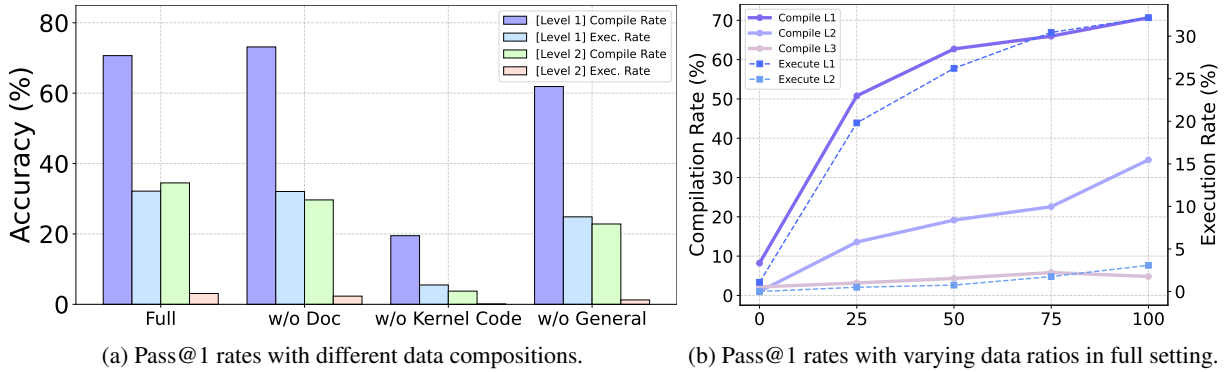


Figure 5: Ablation results of KernelGen-LM-8B on training data composition. (a) Different data composition settings. (b) Full data setting with varying data ratios.

Table 4: RL hyperparameter ablation for 8B.

Negative Strategy	LR	LR Decay	Batch Size	Training Steps	Compile Rate (%)	Exec. Rate (%)
(SFT)	-	-	-	-	60.97	5.18
Compile-pass but execution-fail	1e-6	constant	64	150	44.21	8.31
Compile-fail	1e-6	cosine	64	150	54.03	9.49
Compile-fail	1e-6	constant	64	150	32.00	6.10

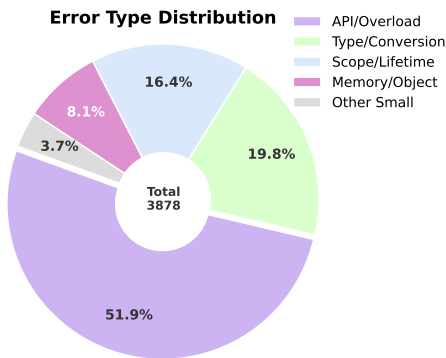


Figure 6: Distribution of kernel generation errors.

5.4 Ablation Analysis of RL

Table 4 shows an ablation study of RL, where negative samples are constructed following the protocol in Sec. 4.2.2. Using compile-pass but execution-fail samples consistently outperforms compile-fail negatives, suggesting that supervision targeting semantic and execution-level errors is more informative than penalizing syntactic failures alone. In addition, a cosine learning-rate schedule yields more stable optimization than a constant schedule.

5.5 Error Analysis

We analyze approximately 4,000 failed kernel generations to identify the dominant failure modes in LLM-based NPU kernel synthesis. Figure 6 shows that API signature and overload mismatches are the most prevalent errors, accounting for 51.9% of

failures, followed by data type and conversion errors (19.8%), variable scope and lifetime issues (16.4%), and memory or object misuse (8.1%). Purely syntactic or structural violations are rare (3.7%), indicating that most generated kernels are syntactically well-formed. Detailed analysis are provided in Appendix C. Overall, failures reflect the constraints of low-level programming, particularly in adhering to API contracts, navigating hardware-specific type systems, and managing cross-component variable visibility.

6 Conclusion

This paper presents *AscendKernelGen*, a unified framework for LLM-based kernel generation under strict NPU hardware constraints. By integrating a reasoning-oriented dataset, domain-adaptive post-training, and a hardware-grounded evaluation benchmark, we systematically bridge the gap between general-purpose code generation and low-level NPU kernel programming. Experiments show that domain-adaptive training significantly improves compilation robustness, functional correctness, and execution efficiency, especially for kernels that general-purpose LLMs fail to handle, and that error-derived supervision and execution-based preference learning are crucial for internalizing hardware-specific constraints.

Limitations

This work studies LLM adaptation for NPU kernel generation using the Ascend platform as a concrete testbed. Although the overall pipeline, including structured reasoning construction, compiler-grounded supervision, and execution-based verification, is conceptually applicable to other accelerator ecosystems, our current instantiation remains platform-specific. AscendC exposes a proprietary and highly structured programming model with explicit memory hierarchy management, asynchronous pipelining, synchronization, and boundary-sensitive arithmetic. As a result, transferring the current system to other hardware backends would require substantial re-collection of domain data, benchmark redesign, and adaptation to different compiler and runtime interfaces.

A second limitation lies in the benchmark design itself. NPUKernelBench is intentionally constructed as a controlled evaluation environment with standardized templates, API descriptions, and strict verification protocols. This design is useful for isolating whether a model has acquired the core reasoning patterns required for hardware programming, but it does not fully capture the broader variability of open-ended operator development in production environments. In particular, real-world deployment often involves less standardized interfaces, more heterogeneous software dependencies, and broader integration challenges beyond kernel logic alone. Therefore, our benchmark should be viewed as measuring low-level hardware reasoning under controlled conditions rather than exhaustively characterizing all aspects of accelerator software development.

In addition, while domain-adaptive training substantially improves compilation success and functional correctness, the current approach does not explicitly optimize for global performance optimality. Passing compilation and correctness checks is only the first step toward deployable kernels; highly efficient kernels also require strong latency optimization, careful tiling strategies, and hardware utilization close to architectural limits. Our current alignment procedure focuses primarily on correctness-oriented supervision, and generalization to more complex or unseen kernel patterns, especially higher-level kernels with long-range dependencies and intricate control flow, remains constrained by the coverage of the training distribution.

Finally, the proposed training and evaluation

pipeline depends on compiler-backed, execution-based verification on real NPU hardware. While this is important for obtaining faithful measurements, it also raises the barrier to entry for researchers who do not have access to comparable devices and software environments. Hardware availability may therefore limit the ease of reproducing the full pipeline end-to-end. Future work could improve accessibility by extending the methodology to other accelerator platforms, incorporating more open evaluation backends, and introducing performance-aware alignment objectives that optimize not only correctness but also runtime efficiency and resource utilization.

Acknowledgments

We would like to express our sincere gratitude to the reviewers for their valuable feedback and constructive comments, which significantly improved the quality of this work. This work is supported by Guangdong S&T Program (Grant No. 2024B0101010003).

References

- Jason Ansel, Edward Z. Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, and 30 others. 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, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024 - 1 May 2024*, pages 929–947. ACM.
- Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. [Program synthesis with large language models](#). *CoRR*, abs/2108.07732.
- Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman P. Amarasinghe. 2019. [Tiramisu: A polyhedral compiler for expressing fast and portable code](#). In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019*, pages 193–205. IEEE.
- Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. 2025. [Kevin: Multi-turn RL for generating CUDA kernels](#). *CoRR*, abs/2507.11948.

- Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. **TVM: an automated end-to-end optimizing compiler for deep learning**. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 578–594. USENIX Association.
- Wentao Chen, Jiace Zhu, Qi Fan, Yehan Ma, and An Zou. 2025. **CUDA-LLM: llms can write efficient CUDA kernels**. *CoRR*, abs/2506.09092.
- DeepSeek-AI. 2025. **Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning**. *CoRR*, abs/2501.12948.
- Juncheng Dong, Yang Yang, Tao Liu, Yang Wang, Feng Qi, Vahid Tarokh, Kaushik Rangadurai, and Shuang Yang. 2025. **STARK: strategic team of agents for refining kernels**. *CoRR*, abs/2510.16996.
- Zacharias V. Fisches, Sahan Paliskara, Simon Guo, Alex Zhang, Joe Spisak, Chris Cummins, Hugh Leather, Gabriel Synnaeve, Joe Isaacson, Aram Markosyan, and Mark Saroufim. 2025. **Kernelllm: Making kernel development more accessible**. Corresponding authors: Aram Markosyan, Mark Saroufim.
- Gemini-Team. 2025. Gemini 2.5 pro model documentation. <https://ai.google.dev/gemini-api/docs/models>. Accessed 2026.
- Xiaodong Gu, Meng Chen, Yalan Lin, Yuhang Hu, Hongyu Zhang, Chengcheng Wan, Zhao Wei, Yong Xu, and Juhong Wang. 2025. **On the effectiveness of large language models in domain-specific code generation**. *ACM Trans. Softw. Eng. Methodol.*, 34(3):78:1–78:22.
- Yue Guan, Yichen Lin, Xu Zhao, Jianzhu Yao, Xinwei Qiang, Zhongkai Yu, Pramod Viswanath, Yufei Ding, and Adnan Aziz. 2026. **Tritongym: A benchmark for agentic LLM workflows in triton GPU code generation**.
- Ping Guo, Chenyu Zhu, Siyuan Chen, Fei Liu, Xi Lin, Zhichao Lu, and Qingfu Zhang. 2025. **Evoengineer: Mastering automated CUDA kernel code evolution with large language models**. *CoRR*, abs/2510.03760.
- Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. **Lora: Low-rank adaptation of large language models**. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.
- Binyuan Hui, Jian Yang, Zeyu Cui, Jiayi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. **Qwen2.5-coder technical report**. *CoRR*, abs/2409.12186.
- Robert Tjarko Lange, Qi Sun, Aaditya Prasad, Maxence Faldor, Yujin Tang, and David Ha. 2025. **Towards robust agentic CUDA kernel benchmarking, verification, and optimization**. *CoRR*, abs/2509.14279.
- Xiaoya Li, Xiaofei Sun, Albert Wang, Jiwei Li, and Chris Shum. 2025. **CUDA-L1: improving CUDA optimization via contrastive reinforcement learning**. *CoRR*, abs/2507.14111.
- Heng Liao, Jiajin Tu, Jing Xia, Hu Liu, Xiping Zhou, Honghui Yuan, and Yuxing Hu. 2021. **Ascend: a scalable and unified architecture for ubiquitous deep neural network computing : Industry track paper**. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2021, Seoul, South Korea, February 27 - March 3, 2021*, pages 789–801. IEEE.
- Cong Liu, Zhong Wang, ShengYu Shen, Jiali Peng, Xiaoli Zhang, ZhenDong Du, and YaFang Wang. 2025. **The chinese dataset distilled from deepseek-r1-671b**. <https://huggingface.co/datasets/Congliu/Chinese-DeepSeek-R1-Distill-data-110k>.
- Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. 2018. **NVIDIA tensor core programmability, performance & precision**. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2018, Vancouver, BC, Canada, May 21-25, 2018*, pages 522–531. IEEE Computer Society.
- Alexander Novikov, Ngân Vu, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. 2025. **Alphaevolve: A coding agent for scientific and algorithmic discovery**. *CoRR*, abs/2506.13131.
- Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. 2025. **Kernelbench: Can llms write efficient GPU kernels?** In *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*, Proceedings of Machine Learning Research. PMLR / OpenReview.net.
- Oded Ovadia, Meni Brief, Rachel Lemberg, and Eitam Sheerit. 2025. **Knowledge-instruct: Effective continual pre-training from limited data using instructions**. *CoRR*, abs/2504.05571.
- Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D. Manning, Stefano Ermon, and Chelsea Finn. 2023. **Direct preference optimization: Your language model is secretly a reward model**. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

- Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. [Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines](#). In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 519–530. ACM.
- Daniel Snider and Ruofan Liang. 2023. [Operator fusion in XLA: analysis and evaluation](#). *CoRR*, abs/2301.13062.
- Qwen Team. 2025. [Qwen3 technical report](#). *CoRR*, abs/2505.09388.
- Philippe Tillet, Hsiang-Tsung Kung, and David D. Cox. 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, MAPL@PLDI 2019, Phoenix, AZ, USA, June 22, 2019*, pages 10–19. ACM.
- voidful. 2025. [reasoning_gemini_300k](#). HuggingFace dataset. https://huggingface.co/datasets/voidful/reasoning_gemini_300k, accessed 2026-04-17.
- Jiin Woo, Shaowei Zhu, Allen Nie, Zhen Jia, Yida Wang, and Youngsuk Park. 2025. [Tritonrl: Training llms to think and code triton without cheating](#). *CoRR*, abs/2510.17891.
- Bartłomiej Wróblewski, Gioele Gottardo, and Anastasios Zouzias. 2025. [Parallel scan on ascend AI accelerators](#). In *2025 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2025 - Workshops, Milano, Italy, June 3-7, 2025*, pages 1290–1292. IEEE.
- Weicheng Xue, Kai Yang, Yongxiang Liu, Dengdong Fan, Pengxiang Xu, and Yonghong Tian. 2024. [Unlocking high performance with low-bit npus and cpus for highly optimized hpl-mxp on cloud brain II](#). In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2024, Atlanta, GA, USA, November 17-22, 2024*. IEEE.
- Devansh Yadav and Shouvick Mondal. 2025. [Evaluating pre-trained large language models on zero shot prompts for parallelization of source code](#). *J. Syst. Softw.*, 230:112543.
- Zijian Zhang, Rong Wang, Shiyang Li, Yuebo Luo, Mingyi Hong, and Caiwen Ding. 2025. [Cudaforge: An agent framework with hardware feedback for CUDA kernel optimization](#). *CoRR*, abs/2511.01884.
- Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, Peng Di, Kun Zhang, and Xuefeng Jin. 2021. [AKG: automatic kernel generation for neural processing units using polyhedral transformations](#). In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 1233–1248. ACM.
- Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. [Anso: Generating high-performance tensor programs for deep learning](#). In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 863–879. USENIX Association.

Contents	
1 Introduction	1
2 Related Works	2
3 Challenges in NPU Kernel Generation	3
4 AscendKernelGen	3
4.1 Ascend-CoT Dataset	3
4.1.1 Documentation-Based CoT Supervision	4
4.1.2 Code-Centric CoT Super- vision	4
4.1.3 General CoT Supervision .	5
4.2 KernelGen-LM: Domain-Adaptive Training for NPU Kernels	5
4.2.1 Error-Derived Supervised Fine-Tuning	5
4.2.2 Reinforcement Learning with Execution-Based Preferences	5
4.3 NPUKernelBench: Evaluation of NPU Kernel Generation	6
5 Experiment	6
5.1 Experimental Setups	6
5.2 Main Results	7
5.2.1 Compilation and Execution	7
5.2.2 Speedup	7
5.3 Ablation Analysis of SFT	7
5.3.1 Model Scale Sensitivity .	7
5.3.2 Comparison of Fine- tuning Strategies	7
5.3.3 Training Data Composi- tion Analysis	8
5.4 Ablation Analysis of RL	9
5.5 Error Analysis	9
6 Conclusion	9
A Kernel Task Categorization in NPUKernelBench	12
B Detail Configurations in Experiment	12
C Detailed Error Taxonomy	12
D Prompts Formulation	13
D.1 Prompt for Documentation-based CoT	13
D.2 Prompt for Kernel-Level CoT . . .	14
D.3 Prompt for Project-Level CoT . .	14
D.4 Prompt for Generating Kernel in NPUKernelBench	14
E CoT Examples	14
E.1 Documentation-based CoT	14
E.2 Kernel-level CoT	14
E.3 Project-level CoT	15
E.4 API-level Debug CoT	15
E.5 Kernel-level Debug CoT	15

A Kernel Task Categorization in NPUKernelBench

Kernel tasks are categorized into three levels according to their inherent computational structure (Table 5). Level 1 tasks consist of simple element-wise or arithmetic operations with linear data flow. Level 2 tasks correspond to common neural network operators with structured computation and local dependencies. Level 3 tasks include operators with global dependencies, iterative computation, or dynamic control flow, such as Gemm, TopK, and attention-related kernels. These tasks place increasing demands on parallel reasoning, memory hierarchy management, and control logic synthesis.

Table 5: Categorization of kernel tasks in NPUKernelBench.

Level	Static-shape	Dynamic-shape	Categories	Representative	Tasks
Level 1	Can linear data flow be mapped to fixed-size loops with deterministic memory access?	Can element-wise logic remain correct under runtime-determined tensor shapes?	Comparison	Add, Equal	3
			Condition	IsFinite, IsInf	3
			Index	GatherV3, ScatterList	2
			Math	AddCustom, Sqrt	26
			TensorCreation	Arrange, Eye	3
Level 2	Can structured computation exploit fixed tiling and local data reuse?	Can kernels generalize correctly across varying input sizes?	Activation	Gelu, MulSigmoid	14
			Foreach	ForeachAbs, ForeachSqrt	53
			Linalg	Cos, Matmul	2
			Loss	CrossEntropyLoss, MseLoss	4
			Mask	Tril, Triu	3
			Norm	AddLayerNorm, RmsNorm	16
			Optim	ApplyAdamWV2	2
			Reduce	MulSigmoidMulAddCustom	4
			TensorMove	ExpandV2, ReverseSequence	5
			Level 3	Can complex kernels with global dependencies and non-trivial execution logic be generated correctly?	Can kernels support dynamic shapes and control flow in complex execution scenarios?
Sort	TopKV3	1			
			Total	16	158

B Detail Configurations in Experiment

As summarized in Table 6, we employ different large language models at each stage according to their respective roles. For dataset construction, DeepSeek-R1 (DeepSeek-AI, 2025) is used for Ascend Documentation-based CoT and Kernel-level CoT construction (Secs. 4.1.1 and 4.1.2). To support Project-level CoT construction that requires reasoning over complete kernel projects with multiple mutually referenced source files, we adopt Gemini 2.5 Pro (Gemini-Team, 2025) (Sec. 4.1.2), which owing to its stronger support for multi-file code understanding and cross-file dependency reasoning. Within the error-derived supervision in SFT (Secs. 4.2.1), DeepSeek-Reasoner is further leveraged to iteratively refine and correct the generated reasoning chains.

C Detailed Error Taxonomy

We provide a detailed taxonomy of kernel generation failures observed in our evaluation. Each category summarizes a recurring failure pattern; representative examples and statistics are shown below.

- **API Signature & Overload Errors:** These errors arise when a generated function call does not match any valid API signature or overload defined in the programming interface. Common causes include incorrect argument ordering, mismatched parameter types, missing required parameters, or confusion between similarly named APIs with subtle semantic differences. This category reflects the models difficulty in aligning high-level task intent with rigid, hardware-specific API contracts.
- **Data Type & Conversion Errors:** This category includes the use of invalid or non-existent data types (e.g., `bool_t`), as well as illegal or unsafe type conversions between incompatible types (e.g., mixing `float` and `double` constants). Such errors often stem from implicit assumptions carried over from high-level programming languages, which do not hold in low-level kernel development where type rules are strict and hardware-dependent.

Table 6: Overview of models and configurations used for dataset construction and two-stage training.

Stage	Model / Method	Role	Description
Dataset Construction	DeepSeek-R1	Ascend Documentation-based CoT & Kernel-level CoT	Document-based data generation and kernel-level CoT construction
	Gemini 2.5 Pro	Project-level CoT	Reasoning over complete kernel projects with multiple mutually referenced source files
	DeepSeek-Reasoner	Error-derived Supervision Refinement	Iterative correction and refinement of generated reasoning chains
Supervised Fine-tuning	Qwen3-32B	Backbone Model	Backbone model for supervised fine-tuning
	Qwen3-1.7B / 4B / 8B / 14B / 32B / Coder-30B-A3B-Instruct	Scaling Study	Supervised fine-tuning across different model scales and architectures
	Training Configuration	Full FT / LoRA	Learning rate 1.25×10^{-6} (Full FT) vs. 1.25×10^{-5} (LoRA), cosine decay, warm-up ratio 0.01, micro/global batch size 1/128, weight decay 1×10^{-1} , gradient clipping 1.0.
Reinforcement Learning	DPO	Optimization Objective	Direct Preference Optimization with $\beta = 0.1$
	Training Configuration	Hyperparameters	Learning rate 1×10^{-6} , cosine decay, warm-up ratio 0.1, global batch size 64, 150 training iterations
	Preference Construction	Ablation Analysis	Analysis of different positive and negative sample selection strategies

- **Variable Scope & Lifetime Errors:** These errors occur when variables are used outside their valid scope or before being properly defined. Typical examples include references to undeclared symbols (e.g., `M_PI`) or misuse of variables that are not visible within the current compilation unit or kernel context. This category highlights challenges in consistently reasoning about variable visibility and lifetime across host-side logic, kernel code, and auxiliary configuration structures.
- **Memory & Object Usage Errors:** Errors in this category are related to incorrect usage of core objects such as `TPipe`, `LocalTensor`, and `GlobalTensor`. They include invalid member function invocations and incorrect memory address computations. While less frequent, such errors can lead to severe runtime failures or undefined behavior.
- **Syntax & Structure Errors:** These errors stem from violations of C++ syntax or incomplete kernel class definitions. Examples include missing required methods (e.g., improperly defined or absent `Init` functions), mismatched braces, or malformed template instantiations. These errors are typically associated with incomplete code generation rather than deeper semantic misunderstandings.
- **Macro & Preprocessing Errors:** This category includes errors caused by incorrect macro definitions or improper usage of tiling-related preprocessing directives. These issues often arise from incorrect assumptions about compile-time constants or conditional compilation logic embedded in kernel templates.

D Prompts Formulation

D.1 Prompt for Documentation-based CoT

The prompt to generate Documentation-based CoT in Sec. 4.1.1 is shown in Table 8. Specifically, we instruct the model to act as an Ascend C expert, transforming raw documentation and Q&A pairs into

reasoning-rich traces. The prompt enforces a deductive process where the model must synthesize the provided reference materials with domain logic, explicitly articulating the derivation path from the initial question to the final solution. Instead of merely summarizing text, the model is required to generate a natural, step-by-step thought process analyzing the problem scope and applying necessary constraints before presenting the result. This ensures the resulting dataset captures the intrinsic problem-solving logic of NPU programming, structured into distinct thinking and answering components.

D.2 Prompt for Kernel-Level CoT

To generate a question for each kernel file, we first analyze the internal call relationships to identify all dependent files. Based on this analysis, we reorganize the target file together with its dependencies into a single, self-contained code snippet that can be compiled without errors. This consolidated snippet is then provided along with its dependency code as input to the LLM, together with the prompt specified in Table 9, to guide the LLM in generating a question that accurately reflects the code's functionality.

D.3 Prompt for Project-Level CoT

To effectively guide the LLM in generating a complete kernel project, it is necessary to construct CoT at the project level for supervised finetuning. Consequently, prior to formulating the CoT, we first design the prompt that will serve as its structural basis. In particular, the prompt is composed of four elements: the instruction text, kernel source code, test cases, and NPU hardware configuration. These components are then concatenated in a predefined order, thereby forming a single, coherent input file that is subsequently provided to the model. The full prompt specification is presented in Table 10.

D.4 Prompt for Generating Kernel in NPUKernelBench

As illustrated in Table 11, the prompt provided for generating kernel code in NPUKernelBench integrates three complementary components: (1) task specification via API descriptions, (2) structural scaffolding through host and kernel code templates, and (3) role-based and formatting constraints. Together, these components provide sufficient semantic grounding and structural guidance for generating compilable NPU code.

E CoT Examples

E.1 Documentation-based CoT

To equip the model with comprehensive Ascend C domain expertise, we construct a specialized dataset utilizing three core references: the *Operator Development Guide*, *API Documentation*, and *Best Practices*. Adopting the Knowledge-Instruct framework, we transform these resources into high-quality instruction data rather than relying on raw unsupervised pre-training, effectively enhancing knowledge retention while mitigating hallucinations. The construction process is divided into two specialized pipelines: Basic Knowledge QA for conceptual understanding and API Call QA for programming proficiency. For basic knowledge, we employ a "generate-retrieve-answer" workflow, where questions are synthesized to cover all technical points, followed by fine-grained RAG retrieval to generate reasoning-rich answers. For API mastery, we address the challenge of unformatted online documentation by developing an intelligent scraping tool that repairs formula images via OCR and LLM-based semantic correction. These refined documents are then processed through structured prompts to create Documentation QA pairs that emphasize template usage and deep reasoning. A concrete example of the generated Ascend C CoT data is presented in Table 12.

E.2 Kernel-level CoT

After constructing the QA pairs from Table 9, we feed the constructed QA pairs into the LLM to generate kernel-level CoT reasoning. Each QA pair produces a corresponding reasoning trace, which is further organized into a unified JSON format for downstream use. Through this pipeline, we obtain over 7k high-quality kernel-level CoT samples, forming a substantial and diverse dataset for supervised training. A full example of the generated CoT is provided in Table 13.

E.3 Project-level CoT

After constructing the prompt, we feed it to the LLM to generate the project-level CoT. In particular, the thinking component encompasses both the design of host-side tiling strategies and the implementation and pipeline design on the kernel side. Moreover, we also output a JSON summary containing all tiling parameter values following the generated thought chain. This summary is subsequently used to evaluate the quality of the CoT in the following sections. The generated CoT example is presented in Table 14.

E.4 API-level Debug CoT

From Sec.4.2.1, we provide an API-level Debug CoT in Table 15, which illustrates key debugging strategies and thought processes for effectively troubleshooting API calls in the Ascend C environment. This example highlights the critical steps and considerations required for successful debugging, offering valuable insights for LLMs.

E.5 Kernel-level Debug CoT

To address silent numerical inconsistencies (e.g., memory staging or tiling errors) in executable kernels, we utilize ground-truth-guided reconstruction. By pairing failing kernels with verified reference implementations, we generate reconstruction-oriented CoT supervision. This is detailed in Sec.4.2.1, along with the kernel-level debug CoT provided in Table 16.

Table 7: Representative examples of errors in generated kernels, grouped by error type.

Error Type	Example	Description
API Signature and Overloading	Equal	Muls: third argument LocalTensor<float> instead of scalar; Greater: function not declared; Mins: third argument minValLocal LocalTensor<float> instead of float; DataCopy: LocalTensor<float> vs GlobalTensor<half> type conflict; ReduceSum: missing required template parameter pattern.
	Arange	Add: 4 arguments including float16 scalar, but all overloads require LocalTensor and more parameters; Duplicate: 3 arguments (LocalTensor, int, int) provided, but all overloads require 6 arguments or have type conflicts (unsigned int vs int).
Data Type and Casting	Equal	equal: float to bool conversion unsupported by CastIntrinsicsImpl; Duplicate: target yLocal type LocalTensor<bool> conflicts with template float; Sub: input tensors type inference conflict (bool vs float).
	Less	Duplicate used with uint8_t/bool, unsupported types; int16_t to bool conversion missing CastIntrinsicsImpl overload.
Memory and Object Misuse	Icamax	TQue missing GetSize; undefined FLT_MAX; undefined fabs; another TQue missing GetSize; DataCopy: parameter mismatch, expected (GlobalTensor, LocalTensor, int); Init: 4 arguments provided, 5 expected.
	Isamax	Dereferencing uint64_t from LocalTensor::GetPhyAddr(); DataCopy: reinterpret_cast int32_t* to __gm__ int32_t* invalid.
Variable Scope and Lifetime	IsInf	Undeclared identifier 'IsInf'; Undeclared 'ToLocalTensor', likely meant 'LocalTensor', used in And function.
	Ccopy	Undeclared 'complex64' in GlobalTensor/LocalTensor templates and sizeof; Undeclared 'c10', causing template instantiation and sizeof errors.
Syntax and Structural	Snrm2	SyncAll() defined without parameters, called as template SyncAll<true>(); KernelSnrm2 missing member 'tailBlock'; tiling struct registration fails.
	FastGelu-Grad	Variable name starting with digit ('1_702_x'), invalid in C++.
Macro and Preprocessing	ClipByValue	TILING_KEY_IS macro misused; macro expansion may lack semicolon.
	Sasum	'ALIGN_SIZE' ambiguous (user-defined 8 vs AscendC 32); DataCopyPad parameter mismatch; TBuf missing SetFlag member function.

Prompt for generating Documentation-based CoT
<p>You are an Ascend C programming expert using the Huawei Ascend processor. Based on the given question and answer, please generate a detailed thought process. The thought process should demonstrate the complete reasoning path from the question to the answer, including key thinking steps, logical analysis, and derivation.</p> <p>The thought process should:</p> <ol style="list-style-type: none"> 1. Start thinking and reasoning from the question itself, not from the document content; 2. Demonstrate a clear approach to solving the problem; 3. Include necessary analytical steps and intermediate reasoning; 4. Proceed step-by-step to the final answer; 5. Use natural and fluent language; 6. Do not include quotations from the original text, the question, or documents; 7. Demonstrate your expertise and thought process as an Ascend C expert. <p>Example Format:</p> <p>Question: How is double buffering applied in Ascend C? How does it improve computational efficiency?</p> <p>Thought Process: [Yes, the user is asking how double buffering is applied in Ascend C and how it improves computational efficiency. First, I need to review the basic concepts of double buffering.] Double buffering primarily hides the latency between data transfer and computation by alternating the use of two buffers, allowing data transfer and computation to occur in parallel ...]</p> <p>Please ensure that the thought process is logically sound, naturally connecting the question and the answer, allowing readers to understand the derivation process from the question to the answer. While reference documents are provided, please remember:</p> <ul style="list-style-type: none"> - The documents are for knowledge reference only, to help you learn and think. - Your output must stem from the problem itself, not just from the document content. - You should demonstrate your thought process as an AscendC expert, not simply extract information from the document. - You can synthesize document content and your expertise, but it should be presented as a natural thought process. <p>The output should consist of two parts: thought process and answer, in the following format:</p> <pre><think>Thought Process<think> <answer>Answer<answer></pre> <p>Below are some sections from the document that you can refer to to aid your thinking, but these are only for supplementary knowledge. Your answer should be based on your own thinking and reasoning as an AscendC expert, not a simple restatement of the document content: [related_doc_contents]</p>

Table 8: Prompt for generating question in documentation CoT (Sec.4.1.1).

Prompt for generating question in Kernel-level CoT
<p>You are an expert in AscendC programming for the Huawei Ascend processor. Based on the provided answer code, please generate a corresponding request for that code.</p> <p>Background:</p> <p>I maintain an AscendC operator repository that contains many operators, and each operator consists of multiple code files. I am constructing a dataset to train an LLM that can assist in writing AscendC operators. Each file within an operator will serve as a single training sample. I already have the answer code for each sample, but I still need to generate the corresponding request. After I provide you with a piece of answer code, your task is to analyze the code, understand and summarize its purpose, and then generate a request such that the provided code naturally serves as a correct response to that request. The request should be high-level and concise rather than overly detailed.</p> <p>Please output the request in the following format:</p> <pre><request>REQUEST_TEXT</request></pre> <p>Note: Output only the final request no reasoning process.</p> <p>Requirements:</p> <ol style="list-style-type: none"> 1. The request must capture the core functionality and purpose of the code. 2. The request should be clear and concise, avoiding unnecessary complexity. <p>The answer code is provided below:</p> <pre>[... Kernel code ...]</pre>

Table 9: Prompt for generating question in Kernel-level CoT (Sec.4.1.2).

Prompt for generating Project-level CoT in Sec. 4.1.2

Instructions

You are an AscendC expert. Based on the problem, code, and hardware docs, generate a reasoning path that derives the code logic, followed by a precise summary of all structure member values and the tilingKey.

Requirements:

Start reasoning from the problem itself, with reasoning details aligned with the implementation logic;

.....

The chain of thought should be output in the following format:

<think>chain-of-thought</think> .

Test cases

Function Description

Operator Function

Returns a tensor with the same shape as the input, where each element is absolute value of the corresponding input element.

Formula

\$\$

$x = [x_0, x_1, \dots, x_{n-1}] \setminus \setminus$

$y = [y_0, y_1, \dots, y_{n-1}]$

\$\$

\$\$

$y_i = |x_i|, \text{quad } i = 0, 1, \dots, n-1$

\$\$

API Definition

Python API

This operator is implemented in C++ and exposed to Python via PyBind11 as 'kernel_gen_ops.abs_math()':

```
"""python
```

```
def abs_math(tensor):
```

```
    """
```

```
    Custom AbsMath operator.
```

```
    Args:
```

```
        tensor (Tensor): Device-side aclTensor (input x).
```

```
        Supported dtypes: bfloat16, float16, float32, int32, int64, DT_COMPLEX64.
```

```
        Supported format: ND.
```

```
    Returns:
```

```
        Tensor: Device-side aclTensor (output y).
```

```
        Same dtype/format as input. Output shape matches x.
```

```
    """
```

```
"""
```

Usage Example

```
...
```

NPU hardware configuration

The hardware model and specifications we are using are: [Platform Info configuration begin]

```
[
```

```
[version]
```

```
SoC_version=Ascend910_9392
```

```
Short_SoC_version=Ascend910_93 [PS: Ascend910_93 is the NPU's name.]
```

```
.....
```

```
]
```

Kernel source code

```
"""
```

```
#ifndef ABS_TILING_H
```

```
#define ABS_TILING_H
```

```
#include "register/tilingdata_base.h"
```

```
namespace optiling { .....
```

```
#endif // ADD_CUSTOM_TILING_H
```

```
"""
```

```
=====
```

```
"""cpp
```

```
#include "abs_tiling.h"
```

```
#include "register/op_def_registry.h"
```

```
#include "tiling/platform/platform_ascendc.h"
```

```
namespace optiling {
```

```
    .....
```

```
}
```

```
namespace ops {
```

```
class AbsMath : public OpDef {
```

```
    .....
```

```
}; OP_ADD(AbsMath);
```

```
"""
```

```
=====
```

```
"""cpp
```

```
#ifndef ABS_COMMON_H
```

```
#define ABS_COMMON_H
```

```
    .....
```

```
"""
```

```
=====
```

Table 10: An example of an AbsMath kernel prompt for Ascend kernel generation, comprising four components: instruction text, test cases, NPU hardware configuration, and kernel source code.

Prompt for Generating Kernel Code in NPUKernelBench

Instructions

- **Role:** "You are a professional Ascend kernel development engineer."
- **Target Language and API:** "Please use Ascend C for development."
- **Code Style Requirements:** "Ensure high code readability and include necessary comments."
- **Output Format Requirements:** "Generate separate Kernel and Host code: the Kernel defines <kernel_name>_kernel, and the Host declares Operator<OperatorName>Paras and implements the corresponding computation logic."
- **Key Information Reminder:** "Pay attention to handling the tensor data types, shapes, and layouts properly."

API_Desc.md

```
# aclnnBasicMatmul
## Function Description
### Kernel Function
This Ascend C kernel multiplies two 2D matrices, A and B, a core operation in deep learning models such as linear layers and attention mechanisms. It takes two 2D tensors that meet the matrix multiplication rule and outputs their product.
### Computational Formula
Assuming the input tensors are  $A$  ( $m \times k$ ) and  $B$  ( $k \times n$ ). The output tensor  $C$  ( $m \times n$ ) is computed as:  $C_{ij} = \sum_{p=1}^k A_{ip} B_{pj}$ , where  $i$  ranges from 1 to  $m$ , and  $j$  ranges from 1 to  $n$ .  $C_{ij}$  denotes the element in the  $i$ -th row and  $j$ -th column of  $C$ .
### Computation Process and Type Conversion
To maintain numerical precision and prevent overflow in large-scale accumulation, this kernel uses high-precision accumulation:
1. It receives two 'float16' input tensors 'a' and 'b'.
2. During multiply-accumulate, products of the 'float16' inputs are converted to 'float32' and accumulated in 'float32'.
3. After accumulation, a 'float32' result tensor is produced.
4. This 'float32' tensor is then converted back to 'float16' as the final output.
## Interface Definition
### Kernel Prototype Definition Interface
#### Input
- a: Device-side aclTensor corresponding to A in the formula; supports float16, 2D, and ND format.
- b: Device-side aclTensor corresponding to B in the formula; supports float16, 2D, and ND format.
#### Output
- c: Device-side aclTensor corresponding to C in the formula; supports float16, 2D, and ND format.
## Constraints and Limitations
* The input tensors 'a' and 'b' currently support only the 'float16' data type.
* The second dimension (number of columns) of 'a' must be equal to the first dimension (number of rows) of 'b'.
* The input tensors support only the ND data format.
```

Host template

```
#include "register/op_def_registry.h"
#include "tiling/platform/platform_ascendc.h"
namespace optiling {
static ge::graphStatus TilingFunc(gert::TilingContext *context)
{
    context->SetBlockDim(platform_ascendc::PlatformAscendCManager::GetInstance()->GetCoreNumAic());
    return ge::GRAPH_SUCCESS;
} } // namespace optiling
namespace ops {
class BasicMatmul : public OpDef {
public:
    explicit BasicMatmul(const char *name) : OpDef(name)
    {
        this->Input("a")
            .ParamType(REQUIRED)
            .DataType({ge::DT_FLOAT16})
            .Format({ge::FORMAT_ND});
        this->Input("b")
            .ParamType(REQUIRED)
            .DataType({ge::DT_FLOAT16})
            .Format({ge::FORMAT_ND});
        this->Output("c")
            .ParamType(REQUIRED)
            .DataType({ge::DT_FLOAT16})
            .Format({ge::FORMAT_ND});
        this->AICore()
            .SetTiling(optiling::TilingFunc)
            .AddConfig("ascend910_93")
            .AddConfig("ascend910b");
    }; OP_ADD(BasicMatmul);
} // namespace ops
```

Kernel template

```
#include <kernel_operator.h>
using namespace AscendC;
extern "C" __global__ __aicore__ void basic_matmul(GM_ADDR a, GM_ADDR b, GM_ADDR c, GM_ADDR workspace,
GM_ADDR tiling)
{ }
```

Table 11: An example of the structured prompt used for Ascend kernel generation, consisting of three components: API description, host/kernel-side template.

Documentation-based CoT in Sec 4.1.1

```
1 {  
2 "input": "In Ascend C, what is the unit of the return value of the GetBlockSize  
   function? ",  
3 "output": "<think>The user's question is about the return unit of the  
   GetBlockSize function in Ascend C. First, I need to analyze it from the  
   function name and function. 'GetBlockSize' literally translates to 'get block  
   size.' Considering the common memory management needs in AI accelerator  
   programming, we can infer that this function is related to memory block  
   specifications. In heterogeneous computing scenarios, the unit of memory block  
   usually involves bytes or hardware-specific storage units (such as matrix  
   blocks in Tensor Cores). However, considering that Ascend C is a programming  
   language geared towards developers, its API design should conform to general  
   programming habits.\n\nFurther thinking, the parameters and return values of  
   memory allocation functions such as malloc in the C/C++ standard library are  
   all in bytes. This design principle is likely to continue in Ascend." In C++,  
   the memory alignment requirements in the NPU architecture are typically  
   designed based on byte units, such as 128-byte alignment. As a platform  
   information query interface, `GetBlockSize` returning the number of bytes  
   better supports developers in performing precise memory calculations and  
   allocations. Furthermore, the function prototype returns an `int64_t` (64-bit  
   signed integer), a data type choice suitable for storing large memory blocks (e.  
   g., accurately representing GB-level memory). If the unit is a larger unit of  
   measurement (e.g., KB/MB), a smaller numerical range may be used, further  
   supporting the determination of the byte unit. Finally, special unit cases need  
   to be excluded: although AI accelerators have dedicated storage structures (e.  
   g., matrix blocks of Cube cells), `GetBlockSize`, as a general interface,  
  
   should return the basic storage unit. Considering the characteristics of the  
   NPU platform and programming conventions, the unit can be determined to be  
   bytes. The return value of the `GetBlockSize` function is in bytes. This  
   function is used to retrieve a data block on the current NPU platform. The  
   memory size of a block is directly represented by the int64_t value it returns,  
   which indicates the number of bytes occupied by the storage space. This design  
   conforms to the general specifications for memory management in heterogeneous  
   computing, making it easier for developers to perform precise memory allocation  
   and calculations.  
4 }
```

Table 12: An Ascend Documentation-based CoT example for Ascend kernel SFT, comprising a question and its chain-of-thought.

Kernel-level CoT in Sec 4.1.2

```

1 {
2   "qus": "Please implement a utility function for computing the temporary buffer
        size required by the SwiGLU operator. The function should be able to determine
        the maximum and minimum buffer sizes based on the input shape and data type, as
        well as retrieve the buffers factor size. Some dependency code has already
        been provided; please complete the implementation based on the following
        summarized dependencies: [====From swiglu\_tiling.h =====...
        code...===== From /api\_check/host\_apicheck.h =====...code
        ...===== From impl/host\_log.h =====...code...]"
3   "cot": "To solve this problem, I need to implement a utility function that
        computes the temporary buffer size required by the SwiGLU operator. The
        function must determine the maximum and minimum buffer sizes based on the input
        shape and data type, as well as return the buffer-size factor. According to
        the provided dependency code, basic interfaces and parameter-validation
        utilities already exist, so my task is to implement the core logic. Since
        SwiGLU requires intermediate results, its temporary-space needs vary across
        data types, and the Ascend processor also imposes alignment constraints,
        motivating the use of a minimum repeat-unit size. My design sets different
        multipliers for half-precision and single-precision data, defines a 256-byte
        minimum repeat unit, and validates both shape and datatype. The maximum buffer
        size is computed as the multiplier times the larger of the input-tensor byte
        size and the repeat-unit size, while the minimum buffer size uses only the
        repeat-unit size. The factor-size interface simply returns the chosen
        multiplier, enabling callers to compute the exact temporary-space requirement
        dynamically."
4   "ans": "/* !\n * \\file swiglu_tiling.cpp\n * \\brief\n */\n# include \"lib/
        activation/swiglu_tiling.h\"\n\n#include <stdint>\n#include <set>\n\n#include
        \"graph/tensor.h\"\n#include \"impl/host_log.h\"\n#include \"impl/api_check/
        host_apicheck.h\"\n\nnamespace AscendC {\nnamespace {\nconstexpr uint32_t
        SWIGLU_HALF_TMP_SIZE = 6;\nconstexpr uint32_t SWIGLU_FLOAT_TMP_SIZE = 0;\n
        constexpr uint32_t SWIGLU_ONE_REPEAT_BYTE_SIZE = 256;\nstatic constexpr
        uint32_t SWI_GELU_HALF_SIEZ = 2;\nstatic constexpr uint32_t SWI_GELU_FLOAT_SIEZ
        = 4;\nstatic const std::set<uint32_t> SUPPORT_TYPESIZE = { SWI_GELU_HALF_SIEZ,
        SWI_GELU_FLOAT_SIEZ};\nstatic constexpr const char SWI_GELU_GET_MAX_MIN[] = \"
        GetSwiGLUMaxMinTmpSize\";\nstatic constexpr const char SWI_GELU_GET_TMP_BUFFER
        [] = \"GetSwiGLUTmpBufferFactorSize\";\n\ninline uint32_t GetSwiGLUMaxTmpSize(
        const uint32_t inputSize, const uint32_t typeSize)\n{\n    const uint32_t
        calcPro = (typeSize == sizeof(float)) ? SWIGLU_FLOAT_TMP_SIZE :
        SWIGLU_HALF_TMP_SIZE;\n    return calcPro * std::max(inputSize * typeSize,
        SWIGLU_ONE_REPEAT_BYTE_SIZE);\n}\n\ninline uint32_t GetSwiGLUMinTmpSize(const
        uint32_t typeSize)\n{\n    const uint32_t calcPro = (typeSize == sizeof(float))
        ? SWIGLU_FLOAT_TMP_SIZE : SWIGLU_HALF_TMP_SIZE;\n    return calcPro *
        SWIGLU_ONE_REPEAT_BYTE_SIZE;\n}\n} // namespace\n\nvoid GetSwiGLUMaxMinTmpSize(
        const ge::Shape& srcShape, const uint32_t typeSize, uint32_t& maxValue,\n
        uint32_t& minValue, const bool isReuseSource)\n{\n    HighLevelApiCheck::
        SrcShapeSizeVerifyingParameters<SWI_GELU_GET_MAX_MIN>(srcShape.GetShapeSize(),
        typeSize);\n    HighLevelApiCheck::TypeSizeVerifyingParameters<
        SWI_GELU_GET_MAX_MIN>(typeSize, SUPPORT_TYPESIZE);\n    HighLevelApiCheck::
        IsReuseSourceVerifyingParameters<SWI_GELU_GET_MAX_MIN>(isReuseSource);\n
        const uint32_t inputSize = srcShape.GetShapeSize();\n    ASCENDC_HOST_ASSERT(
        inputSize > 0, return, \"Input Shape size must be greater than 0.\");\n\n
        minValue = GetSwiGLUMinTmpSize(typeSize);\n    maxValue = GetSwiGLUMaxTmpSize(
        inputSize, typeSize);\n}\n\nvoid GetSwiGLUTmpBufferFactorSize(const uint32_t
        typeSize, uint32_t &maxLiveNodeCount, uint32_t &extraBuffer)\n{\n
        HighLevelApiCheck::TypeSizeVerifyingParameters<SWI_GELU_GET_TMP_BUFFER>(
        typeSize, SUPPORT_TYPESIZE);\n    extraBuffer = 0;\n    maxLiveNodeCount = (
        typeSize == sizeof(float)) ? SWIGLU_FLOAT_TMP_SIZE : SWIGLU_HALF_TMP_SIZE;\n}
        } // namespace AscendC"
5 }

```

Table 13: A SwiGLU kernel QAT example for Ascend kernel SFT, consisting of three parts: question, CoT, and answer.

Project-level CoT in Sec 4.1.2

```
1 {
2   "input": "Prompt in Table 9.",
3   "output": "<think>\nMy task is to implement an `AbsMath` operator for the Ascend
  processor. The operator's function is simple: take the absolute value of each
  element of the input tensor, i.e., `y = |x|`.\n\n First, let's analyze its
  characteristics. It is a typical element-wise operation with simple logic, no
  complex dependencies, and no matrix multiplication. Therefore, it is vector-
  intensive, meaning on the Ascend processor, I should mainly use the AI Vector (
  AIV) units rather than AI Cube (AIC) units.\n\nThe hardware model is `
  Ascend910_93` with `vector_core_cnt=48` Vector cores. Each core has a Unified
  Buffer (UB) of 196,352 bytes. UB size limits the amount of data that can be
  processed at once. The input is `shape=[1]`, `dtype=float16`. Though very small
  , the implementation (especially tiling) must handle arbitrary sizes. Based on
  this, the implementation is divided into Host-side Tiling logic and Device-side
  Kernel.\n\n**I. Host-side Tiling Strategy Design**\n\nThe core goal of Tiling
  is to decompose the entire computation task into small tiles so that it can be
  processed in parallel on multiple cores and ensure that each data block can be
  loaded into UB. \n\n1. **Data Type and Computation Selection**:\n The `AbsMath`
  operator needs to support multiple data types, each with different methods for
  calculating its absolute value. To differentiate these cases on the kernel
  side, I need to design a `tilingKey`. The host side sets this `tilingKey` based
  on the input data type, and the kernel side uses it to call the corresponding
  processing logic. * For `float16`, `bfloat16`, and `float32` types, the `Abs`
  instruction can be used directly, and I have assigned a `tilingKey` for this. *
  For `int32`, the absolute value needs to be efficiently obtained using the
  bitwise operation `(x ^ (x >> 31)) - (x >> 31)`, which corresponds to a
  separate `tilingKey`. * For `int64`, the bitwise operation `(x ^ (x >> 63)) - (
  x >> 63)` is also used, and a `tilingKey` is also assigned ...
4 }
```

Table 14: A project-level CoT example for Ascend kernel SFT, comprising a question and its CoT.

API-level debug CoT in Sec. 4.2.1

```

1 {
2 "input": "I'm developing an Ascend C operator to implement element-wise addition
of two tensors. In my code, I defined two LocalTensor: one to store the integer
result and the other for temporary floating-point calculations. Then I called
the `Add` function to perform the addition operation, but encountered a
compilation error.\n\nHere is my code snippet:\n```\ncpp\nAscendC::LocalTensor<
float> workLocal = tempBuf.Get<float>(this->tileLength);\nAscendC::LocalTensor<
int32_t> zLocal = outQueueZ.AllocTensor<int32_t>();\n\n// Trying to add
workLocal to zLocal\nAscendC::Add(zLocal, zLocal, workLocal, 1);\n```\n\nThe
error message is as follows:\n```\nerror: no matching function for call to 'Add
'\n AscendC::Add(zLocal, zLocal, workLocal, 1);\n ^~~~~~\n... (Reason
for candidate function being ignored: deduced conflicting types for parameter '
T' ('int' vs. 'float'))\n```\n\nWhy does this type conflict error occur? How
should I correctly call the `Add` function to solve this problem?",
3
4 "output": "<think>\nLet me analyze this compilation error carefully. The error
message clearly states \"deduced conflicting types for parameter 'T' ('int' vs.
'float')\", which means the compiler encountered a type conflict when trying to
deduce the template parameter T.\n\nLet me look at the user's code: they
defined two LocalTensor, zLocal is of type int32_t, and workLocal is of type
float, and then directly called Add(zLocal, zLocal, workLocal, 1). I
immediately realized the problemAdd is a template function, which requires all
LocalTensor involved in the computation to have the same data type. This is a
basic design principle of Ascend C vectorized computation, because the hardware
needs to process data of the same type to achieve efficient parallelism.\n\n
Specifically, when the compiler sees Add(zLocal, zLocal, workLocal, 1), 1)
During this call, it needs to deduce the template parameter T. However, zLocal
is a LocalTensor<int32_t>, and workLocal is a LocalTensor<float>. The compiler
cannot determine whether T should be int32_t or float, which causes a conflict.
I remember the prototype of the Add function is roughly template<typename T>
void Add(LocalTensor<T> dst, LocalTensor<T> src0, LocalTensor<T> src1, int32_t
count), and the template parameter T of all LocalTensors must be consistent.
Therefore, the root of the problem is clear: the user is trying to add a float
type workLocal to an int32_t type zLocal, which violates the Add function's
syntax. API type consistency requirements. To solve this problem, users need to
ensure that all tensors use the same data type, such as converting them all to
float or all to int32_t, depending on their computational needs. If the
business logic requires mixed-precision computation, they may need to perform
explicit type conversion first. The problem you encountered is because the `Add
` API requires all input and output tensors to have the same data type. In your
code, the types of `zLocal` (int32_t) and `workLocal` (float) do not match,
causing template inference to fail. The correct approach is to ensure that all
tensors involved in the computation have consistent data types. Below is the
official example call for \"computing the first n data points of a tensor,\"
demonstrating how to correctly use the 4-parameter version of the `Add`
function: ```\ncpp` `AscendC::Add(dstLocal, src0Local, src1Local,` ` 512);`\n```\n\n
In this example, `dstLocal`, `src0Local`, and `src1Local` must be `LocalTensor
`s of the same data type, and the last parameter specifies the number of
elements to be computed. You need to adjust the tensor definitions to ensure
type consistency before calling the `Add` function."
5 }

```

Table 15: An API-level debug CoT example for Ascend kernel SFT, comprising a question and its CoT.

Kernel-level debug CoT in Sec. 4.2.1

```
1 {
2 Input: \n\n### [Problem Description]\n\nYou are an expert in Ascend C programming.
   Your task is to generate Ascend C code that meets specific task requirements
   based on the provided documentation. [Task Background] I will provide three key
   files describing the operators to be implemented. Please read each one
   carefully: 1. api_desc.md - Contains the interface description and formula
   definition of the operators to be implemented, including the functional
   requirements and mathematical expressions. 2. test_cases.csv - Sets the key
   datatype and shape information for the operators to be implemented, for more
   appropriate tiling design and performance optimization. 3. hardware.txt -
   Provides the hardware model and specifications for running the operators to be
   implemented. [Output Format Requirements] - Output both the thought process and
   the answer. The thought process should be analyzed and deduced reasonably. The
   answer must strictly follow the provided template format. - Do not modify
   existing function names, class definitions, template parameters, namespaces,
   etc. - Ensure that the code in the answer section is within XML tags, with one
   tag block per file. Ensure the tiling structure is correctly registered
   according to the operator reference example, and design the tiling parameters
   and values based on the input shape information.\n\nKey Points for Code
   Implementation\n When implementing the code, please ensure the following:\n1.
   Do not use GET_TILING_DATA to obtain tiling; instead, refer to the
   TilingDataDef syntax in the example to obtain tiling.\n2. The code
   implementation should only meet the input information required by the current
   test_cases.csv. Functions in api_desc.md that are irrelevant to the current
   input information do not need to be implemented.\n3. Pay attention to the
   hardware specifications for block design and transfer, and avoid internal
   address out-of-bounds issues (e.g., the ub_size range needs to be considered
   when calculating the AIV operator).\n4. Pay attention to the strictness and
   correctness of the syntax. Check repeatedly during the generation process to
   avoid any undefined variables and classes, and ensure that the code is
   executable and functionally correct.\n5. Do not use names containing DTYPE,
   such as DTYPE_X and DTYPE_Y, to refer to data types in the code.\n6. Pay
   attention to Ascend. The official C definition for the bfloat16 type is
   bfloat16_t, not bfloat16. 7. Note that the official Ascend C definition for the
   float16 type is float16_t, not float16."
3 "output": "<think>\n</think>\n**1. Error Analysis:**\n\nSeveral serious problems
   in the flawed implementation lead to precision errors:\n\n1. **Computational
   Logic Problems**:\n- The complex multiplication implementation is completely
   flawed. The correct complex multiplication formula should be  $(a+bi)(c+di) = (ac-bd) + (ad+bc)i$ , but the flawed implementation attempts to achieve this
   through incorrect indexing operations, resulting in chaotic computational logic
   .\n- When calculating the real part, resultLocal[this->blockLength] is
   incorrectly used as temporary storage, but this area is actually used to store
   the imaginary part data, causing data overwriting.\n- When calculating the
   imaginary part, resultLocal[this->blockLength * 2] is incorrectly used, which
   is clearly out of bounds because the tensor size is only blockLength * 2.\n
   2. **Ub Out of Bounds Problem**:\n- Directly using resultLocal[this->
   blockLength * 2] Accessing data outside the allocated UB buffer range will
   lead to memory access errors and unpredictable computation results. \n\n3. **
   Improper API Use**:\n - Pipeline design error: The Process() function does
   not implement a true pipeline loop, only performing one copy and computation,
   failing to process the entire data block. \n - Data alignment requirements were
   not considered; although the data volume in the test case was perfectly
   aligned, the code did not explicitly handle alignment. \n\n4. **Disorganized
   Index Management**:\n - Incorrectly assumed that the real and imaginary parts
   could be accessed through simple pointer offsets, failing to correctly
   understand the interleaved storage format of complex data in memory. \n\n**2.
   Revised Thought Chain and Code:** \n<think> \n1. **Task Analysis**:\n -
   Implement pointwise multiplication of complex matrices, with input and output
   of COMPLEX64 (each complex number consists of two floats). \n - The matrix
   shape is 48E128, with a total of 6144 complex elements. \n - 1. **Design
   Approach**:\n Utilizes 48 AIV cores, each processing one row of data (128 complex
   numbers). 2. **Design Concept**:\n - Employs a simple row-level parallel
   strategy, with each core processing one row of data. - Due to the small data
   volume (256 float elements per row) ..."
4 }
```

Table 16: A kernel-level debug CoT example for Ascend kernel SFT, comprising a question and its CoT.