

# AscendKernelGen: LLM-Driven Kernel Generation for NPUs

Anonymous ACL submission

## 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](#).

## 1 Introduction

As deep learning workloads scale, modern AI systems increasingly rely on domain-specific accelerators to sustain computational efficiency and throughput. 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. Nevertheless, practical performance still depends heavily on the quality of the underlying compute kernels.

However, developing high-performance NPU kernels remains challenging and labor-intensive.

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%                                     |

Unlike general-purpose programming, it typically relies on vendor-specific Domain-Specific Languages (DSLs) (e.g., AscendC) that demand explicit control over memory hierarchies (global vs. on-chip), data tiling strategies, asynchronous pipelines for vector (SIMD) and matrix (Cube) units. This complexity renders manual development time-consuming and error-prone, severely limiting the scalability of NPU-based solutions.

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 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 frame-

|     |  |  |     |
|-----|--|--|-----|
| 074 | work for low-level NPU kernel development. The       | execution feedback (Baronio et al., 2025; Li           | 123 |
| 075 | core idea of AscendKernelGen is to bridge the        | et al., 2025; Woo et al., 2025) and agentic frame-     | 124 |
| 076 | gap between general-purpose LLMs and hardware-       | works (Guo et al., 2025; Dong et al., 2025; Zhang      | 125 |
| 077 | specific kernel programming through domain-          | et al., 2025). Despite these advances, existing        | 126 |
| 078 | adaptive data, training, and evaluation. Specifi-    | work predominantly targets well-documented plat-       | 127 |
| 079 | cally, we introduce a reasoning-oriented dataset to  | forms with abundant training data. LLM-driven          | 128 |
| 080 | expose models to kernel-level design logic, adopt    | kernel generation for emerging NPU platforms           | 129 |
| 081 | domain-adaptive post-training to improve execu-      | with domain-specific programming models remains        | 130 |
| 082 | tion robustness, and establish a rigorous benchmark  | largely underexplored, motivating our work on          | 131 |
| 083 | to systematically assess compilation, correctness,   | domain-adaptive training strategies and compre-        | 132 |
| 084 | and performance. Together, these components en-      | hensive evaluation frameworks.                         | 133 |
| 085 | able reliable and scalable evaluation of LLM-based   |  |     |
| 086 | NPU kernel generation beyond surface-level code      |  |     |
| 087 | validity. Our main contributions are threefold:      |  |     |
| 088 | 1. A reasoning-oriented dataset for NPU ker-         | <b>Traditional Kernel Optimization Approaches.</b>     | 134 |
| 089 | nel generation. We construct <i>Ascend-CoT</i> ,     | Prior to LLMs, kernel optimization relied pri-         | 135 |
| 090 | a dataset curated from real-world AscendC            | marily on compiler-based and auto-tuning meth-         | 136 |
| 091 | kernels that captures structured reasoning pat-      | ods. Auto-tuning frameworks like TVM (Chen             | 137 |
| 092 | terns such as pipeline construction, synchron-       | et al., 2018) and Ansof (Zheng et al., 2020) search    | 138 |
| 093 | ization, and arithmetic logic.                       | over optimization spaces for diverse hardware          | 139 |
| 094 | 2. Domain-adaptive post-training for kernel syn-     | backends, while domain-specific languages such         | 140 |
| 095 | thesis. We propose a post-training strategy          | as Halide (Ragan-Kelley et al., 2013) and Tri-         | 141 |
| 096 | tailored to NPU kernel generation, producing         | ton (Tillet et al., 2019) decouple algorithm spec-     | 142 |
| 097 | <i>KernelGen-LM</i> with significantly improved      | ification from execution scheduling. Compiler          | 143 |
| 098 | compilation success and functional correct-          | pipelines like TensorFlow XLA (Snider and Liang,       | 144 |
| 099 | ness over general-purpose code models.               | 2023) and PyTorch 2.0 (Ansel et al., 2024) trans-      | 145 |
| 100 | 3. A comprehensive benchmark for NPU kernel          | form computation graphs into optimized kernels.        | 146 |
| 101 | evaluation. We introduce <i>NPUKernelBench</i> ,     | More specialized approaches leverage polyhedral        | 147 |
| 102 | which evaluates generated kernels in terms           | compilation (Baghdadi et al., 2019; Zhao et al.,       | 148 |
| 103 | of compilation, correctness, and performance         | 2021) or platform-specific libraries (Markidis et al., | 149 |
| 104 | under both static- and dynamic-shape settings.       | 2018) for particular hardwareworkload combina-         | 150 |
| 105 |  | tions. While achieving high performance, these         | 151 |
|     |  | methods typically require substantial expert knowl-    | 152 |
|     |  | edge, incur high development costs, and offer lim-     | 153 |
|     |  | ited cross-architecture generalization, motivating     | 154 |
|     |  | more flexible learning-based approaches.               | 155 |
| 106 | <b>2 Related Works</b>                               |  |     |
| 107 | <b>LLM-based Code Generation.</b> Large Language     | <b>Evaluation Benchmarks for Kernel Generation.</b>    | 156 |
| 108 | Models have shown strong performance in general-     | Standard benchmarks such as HumanEval (Chen,           | 157 |
| 109 | purpose code generation across mainstream lan-       | 2021) and MBPP (Austin et al., 2021) primarily         | 158 |
| 110 | guages, as demonstrated by systems such as           | target functional correctness, making them insuffi-    | 159 |
| 111 | Codex (Chen, 2021), AlphaCode (Novikov et al.,       | cient for hardware kernel generation, where com-       | 160 |
| 112 | 2025), and Qwen2.5-Coder (Hui et al., 2024). How-    | pilection feasibility and performance are equally      | 161 |
| 113 | ever, their effectiveness degrades substantially on  | important. Hardware-oriented benchmarks, includ-       | 162 |
| 114 | domain-specific tasks (Gu et al., 2025), particu-    | ing KernelBench (Ouyang et al., 2025) and Tri-         | 163 |
| 115 | larly hardware programming that requires explicit    | tonGym (Anonymous, 2025), extend evaluation to         | 164 |
| 116 | memory management and architecture-dependent         | compilation and performance. However, most ex-         | 165 |
| 117 | semantics.   | isting benchmarks require full host-device code        | 166 |
| 118 | Recent work has explored LLM-based hardware          | generation for all tasks, limiting flexibility across  | 167 |
| 119 | kernel generation, primarily for mature ecosys-      | different kernel abstraction levels. Moreover, com-    | 168 |
| 120 | tems like CUDA and TPUs. Approaches range            | prehensive evaluation frameworks tailored to NPU       | 169 |
| 121 | from prompting-based methods (Chen et al., 2025;     | kernel generation remain lacking.                      | 170 |
| 122 | Lange et al., 2025) and supervised fine-tuning (Fis- |  |     |
|     | ches et al., 2025), to reinforcement learning with   |  |     |

171  
172  
173  
174  
175  
176  
177  
178  
179  
180  
181  
182  
183  
184  
185  
186  
187  
188  
189  
190  
  
191  
192  
193  
194  
195  
196  
197  
  
198  
199  
200  
201  
202  
203  
204  
  
205  
206  
207  
208  
209  
210  
211  
212  
  
213  
  
214  
215  
216  
217

### 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 depends on auxiliary parameters such as block indices, tiling factors, and boundary sizes that are computed externally and referenced across multiple pipeline stages. These parameters govern memory access and synchronization behavior, requiring semantic consistency across distant code regions.

**Explicit synchronization reasoning.** Asynchronous pipeline stages communicate through inserted synchronization primitives. Correct execution relies on precise pairing and ordering of these operations to enforce producer-consumer relationships, making correctness dependent on reasoning about dynamic execution ordering.

**Boundary- and layout-sensitive reasoning.** Kernel code must explicitly handle boundary cases through offset computation, masking, and layout transformations tailored to specific execution units. This requires precise arithmetic reasoning over indices and validity conditions, as well as careful management of physical data layouts while preserving logical tensor semantics.

### 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.

As illustrated in Figure 1, the dataset comprises three components: documentation-based Chain-of-Thought (CoT) from AscendC manuals, kernel-centric CoT derived from real-world kernels, and general CoT to facilitate generalization.

##### 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.

In contrast, *project-level CoT* targets industrial-grade operators that interleave host-side tiling logic

218  
219  
  
220  
221  
222  
223  
224  
225  
226  
227  
228  
229  
230  
231  
232  
233  
  
234  
235  
236  
237  
238  
239  
240  
241  
242  
243  
244  
245  
246  
247  
248  
249  
250  
  
251  
252  
253  
254  
255  
256  
257  
258  
259  
260  
261  
262  
263  
264  
265  
266

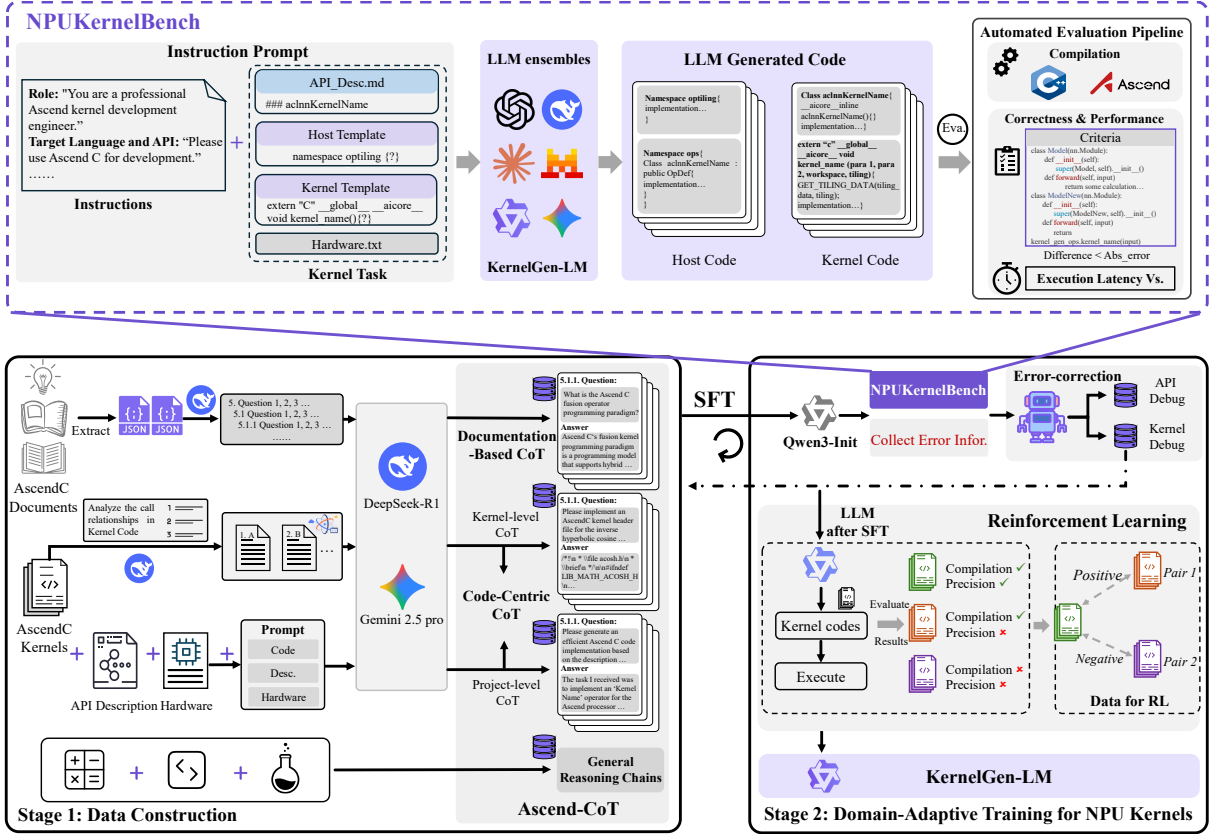


Figure 1: System overview of AscendKernelGen, depicting the data construction, LLM training, and hardware-grounded evaluation pipeline for NPU kernel generation.

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; Hugodonotexit, 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 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*

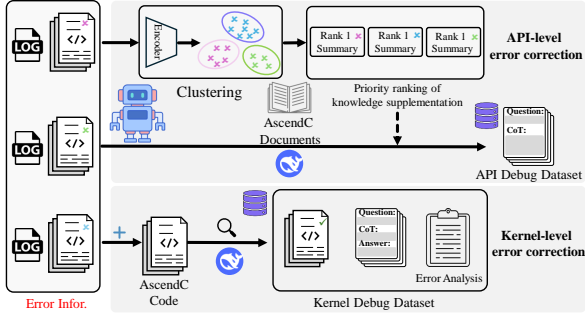


Figure 2: Error-Derived Supervision for API- and Kernel-Level Error Correction.

*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.

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). This mechanism aligns the model with execution-based preferences, effectively filtering out subtle numerical errors that deterministic supervision fails to capture.

#### 4.3 NPUIKernelBench: Evaluation of NPU Kernel Generation

*NPUIKernelBench* is 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, *NPUIKernelBench* 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.

### 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 train-

ing strategy encompassing SFT and RL. For the SFT phase, Qwen3-32B(Qwen-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.

**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 NPUNetBench 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    | <b>71.89</b> | 27.31        | <b>95.18</b> | 67.06        | <b>100</b> | 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        | <b>32.04</b> | 94.62        | <b>77.06</b> | <b>100</b> | <b>88.89</b> | 0.87                 |

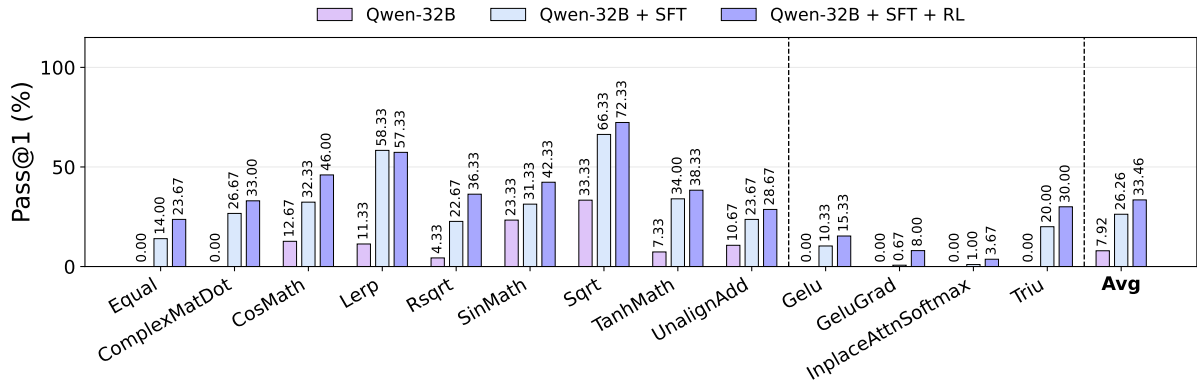


Figure 3: Pass@1 results for representative kernels across training stages.

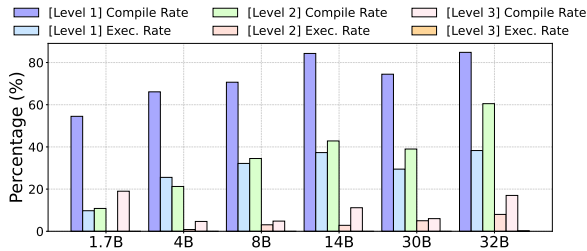


Figure 4: Scaling behavior of KernelGen-LM on NPU kernel generation tasks with increasing difficulty.

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    | <b>55.32</b>     | <b>22.13</b>   | <b>0.95</b>          |

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-

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.

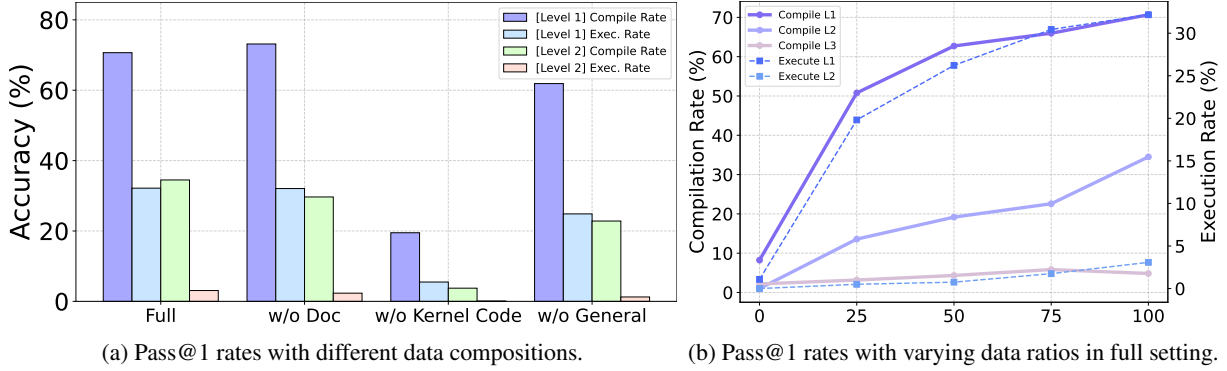


Figure 5: Ablation results of KernelGen-LM-8B on training data composition. (a) Different data composition settings. (b) Full data setting with varying data ratios.

Table 4: RL hyperparameter ablation for 8B.

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

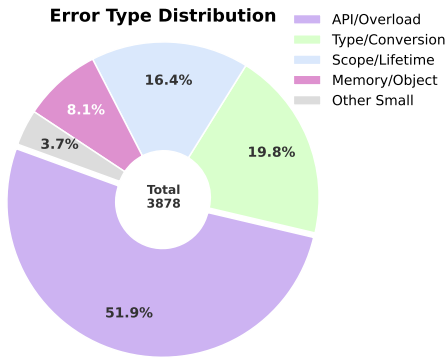


Figure 6: Distribution of kernel generation errors.

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.

## 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

## 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 on real NPU hardware show that domain-adaptive training significantly improves compilation robustness, functional correctness, and execution efficiency, especially for complex 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 beyond syntactic validity.

## 563 Limitations

564 This work focuses on the Ascend NPU and its proprietary programming model. Although the proposed framework is conceptually general, extending it to other accelerator platforms would require additional domain-specific data and adaptation.

565 While domain-adaptive training substantially improves compilation and correctness, the model does not explicitly optimize for global performance optimality, and generalization to unseen kernel patterns remains limited by the training distribution.

574 Finally, the proposed training and evaluation pipeline relies on execution-based verification on real hardware, which may limit accessibility and reproducibility for researchers without comparable NPU resources.

## 579 References

580 Anonymous. 2025. *Tritongym: A benchmark for agentic LLM workflows in triton GPU code generation*. In *Submitted to The Fourteenth International Conference on Learning Representations*. Under review.

584 Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, and 1 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*, pages 929–947.

593 Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

598 Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoab Kamil, and Saman Amarasinghe. 2019. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 193–205. IEEE.

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

610 Mark Chen. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

612 Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, and 1 others.

2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594.

619 Wentao Chen, Jiace Zhu, Qi Fan, Yehan Ma, and An Zou. 2025. Cuda-llm: Llms can write efficient cuda kernels. *arXiv preprint arXiv:2506.09092*.

622 DeepSeek-AI. 2025. *Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning*. *CoRR*, abs/2501.12948.

625 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. *arXiv preprint arXiv:2510.16996*.

629 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.

635 Gemini-Team. 2025. Gemini 2.5 pro model documentation. <https://ai.google.dev/gemini-api/docs/models>. Accessed 2025.

638 Xiaodong Gu, Meng Chen, Yalan Lin, Yuhan 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 Transactions on Software Engineering and Methodology*, 34(3):1–22.

644 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. *arXiv preprint arXiv:2510.03760*.

649 Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, and 1 others. 2022. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3.

653 Hugodonotexit. 2025. R1 dataset collection: Aggregated deepseek-r1 and deepseek-r1-0528 responses. <https://huggingface.co/Hugodonotexit/math-code-science-deepseek-r1-en>.

657 Binyuan Hui, Jian Yang, Zeyu Cui, Jiayi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

662 Robert Tjarko Lange, Aaditya Prasad, Qi Sun, Maxence Faldor, Yujin Tang, and David Ha. 2025. The ai cuda engineer: Agentic cuda kernel discovery, optimization and composition. Technical report, Technical report, Sakana AI, 02 2025.

|     |   |     |
|-----|---|-----|
| 667 | Xiaoya Li, Xiaofei Sun, Albert Wang, Jiwei Li, and Chris Shum. 2025. Cuda-11: Improving cuda optimization via contrastive reinforcement learning. <i>arXiv preprint arXiv:2507.14111</i> .  | 720 |
| 668 |   | 721 |
| 669 |   | 722 |
| 670 |   | 723 |
| 671 | 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 <i>2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)</i> , pages 789–801. IEEE.           | 724 |
| 672 |   | 725 |
| 673 |   |     |
| 674 |   |     |
| 675 |   |     |
| 676 |   |     |
| 677 |   |     |
| 678 | Cong Liu, Zhong Wang, ShengYu Shen, Jialiang Peng, Xiaoli Zhang, ZhenDong Du, and YaFang Wang. 2025. The chinese dataset distilled from deepseek-r1-671b. <a href="https://huggingface.co/datasets/Congliu/Chinese-DeepSeek-R1-Distill-data-110k">https://huggingface.co/datasets/Congliu/Chinese-DeepSeek-R1-Distill-data-110k</a> . | 726 |
| 679 |   | 727 |
| 680 |   | 728 |
| 681 |   | 729 |
| 682 |   |     |
| 683 |   |     |
| 684 | Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. 2018. Nvidia tensor core programmability, performance & precision. In <i>2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW)</i> , pages 522–531. IEEE.  | 730 |
| 685 |   | 731 |
| 686 |   | 732 |
| 687 |   | 733 |
| 688 |   |     |
| 689 |   |     |
| 690 | Alexander Novikov, Ng n V , Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, and 1 others. 2025. Alphaevolve: A coding agent for scientific and algorithmic discovery. <i>arXiv preprint arXiv:2506.13131</i> .                       | 734 |
| 691 |   | 735 |
| 692 |   | 736 |
| 693 |   |     |
| 694 |   |     |
| 695 |   |     |
| 696 | Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher R , and Azalia Mirhoseini. 2025. Kernelbench: Can llms write efficient gpu kernels? <i>arXiv preprint arXiv:2502.10517</i> .  | 737 |
| 697 |   | 738 |
| 698 |   | 739 |
| 699 |   | 740 |
| 700 | Oded Ovadia, Meni Brief, Rachel Lemberg, and Eitam Sheerit. 2025. Knowledge-instruct: Effective continual pre-training from limited data using instructions. <i>arXiv preprint arXiv:2504.05571</i> .   | 741 |
| 701 |   | 742 |
| 702 |   | 743 |
| 703 |   |     |
| 704 | Qwen-Team. 2025. <i>Qwen3 technical report</i> . Preprint, arXiv:2505.09388.  | 744 |
| 705 |   | 745 |
| 706 |   | 746 |
| 707 |   | 747 |
| 708 |   | 748 |
| 709 |   |     |
| 710 |   |     |
| 711 | Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Fr do Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. <i>Acm Sigplan Notices</i> , 48(6):519–530.  | 749 |
| 712 |   | 750 |
| 713 |   | 751 |
| 714 |   | 752 |
| 715 |   | 753 |
| 716 |   | 754 |
| 717 | Daniel Snider and Ruofan Liang. 2023. Operator fusion in xla: analysis and evaluation. <i>arXiv preprint arXiv:2301.13062</i> .   | 755 |
| 718 |   | 756 |
| 719 |   |     |
|     | Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In <i>Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages</i> , pages 10–19.  | 757 |
|     |   | 758 |
|     |   | 759 |
|     |   | 760 |
|     |   | 761 |
|     |   | 762 |
|     |   | 763 |
|     | voidful. 2025. reasoning_gemini_300k. HuggingFace dataset. <a href="https://huggingface.co/datasets/voidful/reasoning_gemini_300k">https://huggingface.co/datasets/voidful/reasoning_gemini_300k</a> , accessed 2025-12-05.   | 764 |
|     |   | 765 |
|     |   | 766 |
|     |   | 767 |
|     | Jiin Woo, Shaowei Zhu, Allen Nie, Zhen Jia, Yida Wang, and Youngsuk Park. 2025. Tritonrl: Training llms to think and code triton without cheating. <i>arXiv preprint arXiv:2510.17891</i> .   | 768 |
|     |   | 769 |
|     |   | 770 |
|     |   | 771 |
|     |   | 772 |
|     |   | 773 |
|     |   | 774 |
|     |   | 775 |
|     |   | 776 |
|     |   | 777 |
|     |   | 778 |
|     |   | 779 |
|     |   | 780 |
|     |   | 781 |
|     |   | 782 |
|     |   | 783 |
|     |   | 784 |
|     |   | 785 |
|     |   | 786 |
|     |   | 787 |
|     |   | 788 |
|     |   | 789 |
|     |   | 790 |
|     |   | 791 |
|     |   | 792 |
|     |   | 793 |
|     |   | 794 |
|     |   | 795 |
|     |   | 796 |
|     |   | 797 |
|     |   | 798 |
|     |   | 799 |
|     |   | 800 |
|     |   | 801 |
|     |   | 802 |
|     |   | 803 |
|     |   | 804 |
|     |   | 805 |
|     |   | 806 |
|     |   | 807 |
|     |   | 808 |
|     |   | 809 |
|     |   | 810 |
|     |   | 811 |
|     |   | 812 |
|     |   | 813 |
|     |   | 814 |
|     |   | 815 |
|     |   | 816 |
|     |   | 817 |
|     |   | 818 |
|     |   | 819 |
|     |   | 820 |
|     |   | 821 |
|     |   | 822 |
|     |   | 823 |
|     |   | 824 |
|     |   | 825 |
|     |   | 826 |
|     |   | 827 |
|     |   | 828 |
|     |   | 829 |
|     |   | 830 |
|     |   | 831 |
|     |   | 832 |
|     |   | 833 |
|     |   | 834 |
|     |   | 835 |
|     |   | 836 |
|     |   | 837 |
|     |   | 838 |
|     |   | 839 |
|     |   | 840 |
|     |   | 841 |
|     |   | 842 |
|     |   | 843 |
|     |   | 844 |
|     |   | 845 |
|     |   | 846 |
|     |   | 847 |
|     |   | 848 |
|     |   | 849 |
|     |   | 850 |
|     |   | 851 |
|     |   | 852 |
|     |   | 853 |
|     |   | 854 |
|     |   | 855 |
|     |   | 856 |
|     |   | 857 |
|     |   | 858 |
|     |   | 859 |
|     |   | 860 |
|     |   | 861 |
|     |   | 862 |
|     |   | 863 |
|     |   | 864 |
|     |   | 865 |
|     |   | 866 |
|     |   | 867 |
|     |   | 868 |
|     |   | 869 |
|     |   | 870 |
|     |   | 871 |
|     |   | 872 |
|     |   | 873 |
|     |   | 874 |
|     |   | 875 |
|     |   | 876 |
|     |   | 877 |
|     |   | 878 |
|     |   | 879 |
|     |   | 880 |
|     |   | 881 |
|     |   | 882 |
|     |   | 883 |
|     |   | 884 |
|     |   | 885 |
|     |   | 886 |
|     |   | 887 |
|     |   | 888 |
|     |   | 889 |
|     |   | 890 |
|     |   | 891 |
|     |   | 892 |
|     |   | 893 |
|     |   | 894 |
|     |   | 895 |
|     |   | 896 |
|     |   | 897 |
|     |   | 898 |
|     |   | 899 |
|     |   | 900 |

764  
765  
766  
767  
768  
769  
770  
771  
772  
773  
774  
775  
776  
777  
778  
779  
780  
781  
782  
783  
784  
785  
786  
787  
788  
789  
790  
791  
792  
793  
794  
795  
796  
797  
798  
799  
800  
801  
802  
803  
804  
805  
806

|   |           |
|---|-----------|
| <b>Contents</b>   |           |
| <b>1 Introduction</b>   | <b>1</b>  |
| <b>2 Related Works</b>  | <b>2</b>  |
| <b>3 Challenges in NPU Kernel Generation</b>                  | <b>3</b>  |
| <b>4 AscendKernelGen</b>                                      | <b>3</b>  |
| 4.1 Ascend-CoT Dataset  | 3         |
| 4.1.1 Documentation-Based CoT Supervision                     | 3         |
| 4.1.2 Code-Centric CoT Supervision                            | 3         |
| 4.1.3 General CoT Supervision                                 | 4         |
| 4.2 KernelGen-LM: Domain-Adaptive Training for NPU Kernels    | 4         |
| 4.2.1 Error-Derived Supervised Fine-Tuning                    | 4         |
| 4.2.2 Reinforcement Learning with Execution-Based Preferences | 5         |
| 4.3 NPUKernelBench: Evaluation of NPU Kernel Generation       | 5         |
| <b>5 Experiment</b>   | <b>5</b>  |
| 5.1 Experimental Setups                                       | 5         |
| 5.2 Main Results  | 6         |
| 5.2.1 Compilation and Execution                               | 6         |
| 5.2.2 Speedup   | 6         |
| 5.3 Ablation Analysis of SFT                                  | 6         |
| 5.3.1 Model Scale Sensitivity                                 | 6         |
| 5.3.2 Comparison of Fine-tuning Strategies                    | 6         |
| 5.3.3 Training Data Composition Analysis                      | 7         |
| 5.4 Ablation Analysis of RL                                   | 8         |
| 5.5 Error Analysis  | 8         |
| <b>6 Conclusion</b>   | <b>8</b>  |
| <b>A Kernel Task Categorization in NPUKernelBench</b>         | <b>11</b> |
| <b>B Detail Configurations in Experiment</b>                  | <b>11</b> |
| <b>C Detailed Error Taxonomy</b>                              | <b>11</b> |
| <b>D Prompts Formulation</b>                                  | <b>12</b> |
| D.1 Prompt for Documentation-based CoT                        | 12        |
| D.2 Prompt for Kernel-Level CoT                               | 13        |
| D.3 Prompt for Project-Level CoT                              | 13        |

|   |           |     |     |
|---|-----------|-----|-----|
| D.4 Prompt for Generating Kernel in NPUKernelBench  | 13        | 807 | 808 |
| <b>E CoT Examples</b>   | <b>14</b> |     | 809 |
| E.1 Documentation-based CoT   | 14        |     | 810 |
| E.2 Kernel-level CoT  | 14        |     | 811 |
| E.3 Project-level CoT   | 14        |     | 812 |
| E.4 API-level Debug CoT   | 14        |     | 813 |
| E.5 Kernel-level Debug CoT  | 14        |     | 814 |
| <b>A Kernel Task Categorization in NPUKernelBench</b>   |           |     | 815 |
| 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.   |           |     | 816 |
|   |           |     | 817 |
|   |           |     | 818 |
|   |           |     | 819 |
|   |           |     | 820 |
|   |           |     | 821 |
|   |           |     | 822 |
|   |           |     | 823 |
|   |           |     | 824 |
|   |           |     | 825 |
|   |           |     | 826 |
|   |           |     | 827 |
|   |           |     | 828 |
|   |           |     | 829 |
| <b>B Detail Configurations in Experiment</b>  |           |     | 830 |
| 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. |           |     | 831 |
|   |           |     | 832 |
|   |           |     | 833 |
|   |           |     | 834 |
|   |           |     | 835 |
|   |           |     | 836 |
|   |           |     | 837 |
|   |           |     | 838 |
|   |           |     | 839 |
|   |           |     | 840 |
|   |           |     | 841 |
|   |           |     | 842 |
|   |           |     | 843 |
|   |           |     | 844 |
|   |           |     | 845 |
|   |           |     | 846 |
| <b>C Detailed Error Taxonomy</b>  |           |     | 847 |
| 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.  |           |     | 848 |
|   |           |     | 849 |
|   |           |     | 850 |
|   |           |     | 851 |
|   |           |     | 852 |

Table 5: Categorization of kernel tasks in NPUIKernelBench.

| 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? | GMM            | Gemm, BasicMatmul         | 17    |
|         |  |   | Sort           | TopKV3                    | 1     |
|         |  |   | Total          | 16                        | 158   |

853 • **API Signature & Overload Errors:** These  
854 errors arise when a generated function call  
855 does not match any valid API signature or  
856 overload defined in the programming inter-  
857 face. Common causes include incorrect argu-  
858 ment ordering, mismatched parameter types,  
859 missing required parameters, or confusion be-  
860 tween similarly named APIs with subtle se-  
861 mantic differences. This category reflects the  
862 models difficulty in aligning high-level task  
863 intent with rigid, hardware-specific API con-  
864 tracts.

865 • **Data Type & Conversion Errors:** This cat-  
866 egory includes the use of invalid or non-  
867 existent data types (e.g., `bool_t`), as well as  
868 illegal or unsafe type conversions between  
869 incompatible types (e.g., mixing `float` and  
870 `double` constants). Such errors often stem  
871 from implicit assumptions carried over from  
872 high-level programming languages, which  
873 do not hold in low-level kernel development  
874 where type rules are strict and hardware-  
875 dependent.

876 • **Variable Scope & Lifetime Errors:** These  
877 errors occur when variables are used outside  
878 their valid scope or before being properly de-  
879 fined. Typical examples include references to  
880 undeclared symbols (e.g., `M_PI`) or misuse of  
881 variables that are not visible within the cur-  
882 rent compilation unit or kernel context. This  
883 category highlights challenges in consistently  
884 reasoning about variable visibility and life-

time across host-side logic, kernel code, and  
885 auxiliary configuration structures. 886

887 • **Memory & Object Usage Errors:** Errors in  
888 this category are related to incorrect usage  
889 of core objects such as `TPipe`, `LocalTensor`,  
890 and `GlobalTensor`. They include invalid  
891 member function invocations and incorrect  
892 memory address computations. While less  
893 frequent, such errors can lead to severe run-  
894 time failures or undefined behavior.

895 • **Syntax & Structure Errors:** These errors  
896 stem from violations of C++ syntax or in-  
897 complete kernel class definitions. Examples  
898 include missing required methods (e.g., im-  
899 properly defined or absent `Init` functions),  
900 mismatched braces, or malformed template  
901 instantiations. These errors are typically asso-  
902 ciated with incomplete code generation rather  
903 than deeper semantic misunderstandings.

904 • **Macro & Preprocessing Errors:** This cat-  
905 egory includes errors caused by incorrect  
906 macro definitions or improper usage of tiling-  
907 related preprocessing directives. These issues  
908 often arise from incorrect assumptions about  
909 compile-time constants or conditional compi-  
910 lation logic embedded in kernel templates.

## D Prompts Formulation 911

### D.1 Prompt for Documentation-based CoT 912

The prompt to generate Documentation-based CoT  
913 in Sec. 4.1.1 in show in Table 8. Specifically, we 914

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 & Kenel-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 $1.25 \times 10^{-6}$ (Full FT) vs. $1.25 \times 10^{-5}$ (LoRA), cosine decay, warm-up ratio 0.01, micro/global batch size 1/128, weight decay $1 \times 10^{-1}$ , gradient clipping 1.0. |
| Reinforcement Learning | DPO   | Optimization Objective                           | Direct Preference Optimization with $\beta = 0.1$   |
|                        | Training Configuration                                    | Hyperparameters                                  | Learning rate $1 \times 10^{-6}$ , 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   |

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 codes 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,

|      |   |      |
|------|---|------|
| 962  | and (3) role-based and formatting constraints. Together, these components provide sufficient semantic grounding and structural guidance for generating compilable NPU code.   | 1010 |
| 963  |   | 1011 |
| 964  |   | 1012 |
| 965  |   | 1013 |
| 966  | <b>E CoT Examples</b>   | 1014 |
| 967  | <b>E.1 Documentation-based CoT</b>  | 1015 |
| 968  | To equip the model with comprehensive Ascend C domain expertise, we construct a specialized dataset utilizing three core references: the <i>Operator Development Guide</i> , <i>API Documentation</i> , and <i>Best Practices</i> . 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. | 1016 |
| 969  |   |      |
| 970  |   |      |
| 971  |   |      |
| 972  |   |      |
| 973  |   |      |
| 974  |   |      |
| 975  |   |      |
| 976  |   |      |
| 977  |   |      |
| 978  |   |      |
| 979  |   |      |
| 980  |   |      |
| 981  |   |      |
| 982  |   |      |
| 983  |   |      |
| 984  |   |      |
| 985  |   |      |
| 986  |   |      |
| 987  |   |      |
| 988  |   |      |
| 989  |   |      |
| 990  |   |      |
| 991  |   |      |
| 992  |   |      |
| 993  |   |      |
| 994  | <b>E.2 Kernel-level CoT</b>   | 1017 |
| 995  | 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.  | 1018 |
| 996  |   |      |
| 997  |   |      |
| 998  |   |      |
| 999  |   |      |
| 1000 |   |      |
| 1001 |   |      |
| 1002 |   |      |
| 1003 |   |      |
| 1004 |   |      |
| 1005 | <b>E.3 Project-level CoT</b>  | 1019 |
| 1006 | 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.   | 1020 |
| 1007 |   |      |
| 1008 |   |      |
| 1009 |   |      |
|      | <b>E.4 API-level Debug CoT</b>  | 1021 |
|      | 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.  | 1022 |
|      |   | 1023 |
|      |   | 1024 |
|      | <b>E.5 Kernel-level Debug CoT</b>   | 1025 |
|      | 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.   | 1026 |
|      |   | 1027 |
|      |   | 1028 |
|      |   | 1029 |
|      |   | 1030 |
|      |   | 1031 |
|      |   | 1032 |
|      |   | 1033 |

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

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.

The thought process should:

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.

Example Format:

Question: How is double buffering applied in Ascend C? How does it improve computational efficiency?

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 ...]

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:

- 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.

The output should consist of two parts: thought process and answer, in the following format:

<think>Thought Process</think>

<answer>Answer</answer>

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 ]

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

### Prompt for generating question in Kernel-level CoT

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.

#### Background:

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.

Please output the request in the following format:

<request>REQUEST\_TEXT</request>

Note: Output only the final request no reasoning process.

#### Requirements:

1. The request must capture the core functionality and purpose of the code.
2. The request should be clear and concise, avoiding unnecessary complexity.

The answer code is provided below:

[ ... Kernel code ... ]

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
        nconstexpr 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\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}\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\nAscendC::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.