AMLA: MUL by ADD in FlashAttention Rescaling

Qichen Liao liaoqichen2@huawei.com
Chengqiu Hu hu.chengqiu@huawei.com
Fangzheng Miao miaofangzheng@huawei.com
Bao Li li.bao@huawei.com
Yiyang Liu liuyiyang16@huawei.com
Junlong Lyu lyujunlong@huawei.com
Lirui Jiang jianglirui1@huawei.com
Jun Wang hwjun.wang@huawei.com
Lingchao Zheng zhenglingchao@huawei.com
Jun Li lijun276@huawei.com
Yuwei Fan* fanyuwei2@huawei.com

Huawei

Abstract

Multi-head Latent Attention (MLA) significantly reduces KVCache memory usage in Large Language Models while introducing substantial computational overhead and intermediate variable expansion. This poses challenges for efficient hardware implementation — especially during the decode phase. This paper introduces Ascend MLA (AMLA), a high-performance kernel specifically optimized for Huawei's Ascend NPUs. AMLA is built on two core innovations: (1) A novel FlashAttention-based algorithm that replaces floating-point multiplications with integer additions for output block rescaling, leveraging binary correspondence between FP32 and INT32 representations; (2) A Preload Pipeline strategy with hierarchical tiling that maximizes FLOPS utilization: the Preload Pipeline achieves Cube-bound performance, while hierarchical tiling overlaps data movement and computation within the Cube core. Experiments show that on Ascend 910 NPUs (integrated in CloudMatrix384 [24]), AMLA achieves up to 614 TFLOPS, reaching **86.8%** of the theoretical maximum FLOPS, outperforming the state-of-the-art open-source FlashMLA implementation, whose FLOPS utilization is up to 66.7%[1] on NVIDIA H800 SXM5 [8]. The AMLA kernel has been integrated into Huawei's CANN and will be released soon.

1 Introduction

Attention mechanisms lie at the heart of the Transformer architecture [20], empowering Large Language Models (LLMs) to model long-range dependencies through explicit token-to-token interactions. As downstream applications increasingly demand ultra-long context windows [22, 18, 14, 16], the computational cost and memory footprint of attention have become critical bottlenecks. While Multi-Query Attention (MQA) [19] and Grouped-Query Attention (GQA) [2] reduce memory pressure via key-value head sharing, Multi-Head Latent Attention (MLA) [9] pushes compression further by

*Corresponding author

[1]The tensor core utilization of FlashMLA is 66.7% of the maximum theoretical peak of H800 SXM5, and 80% of the throttled theoretical peak.

Preprint.

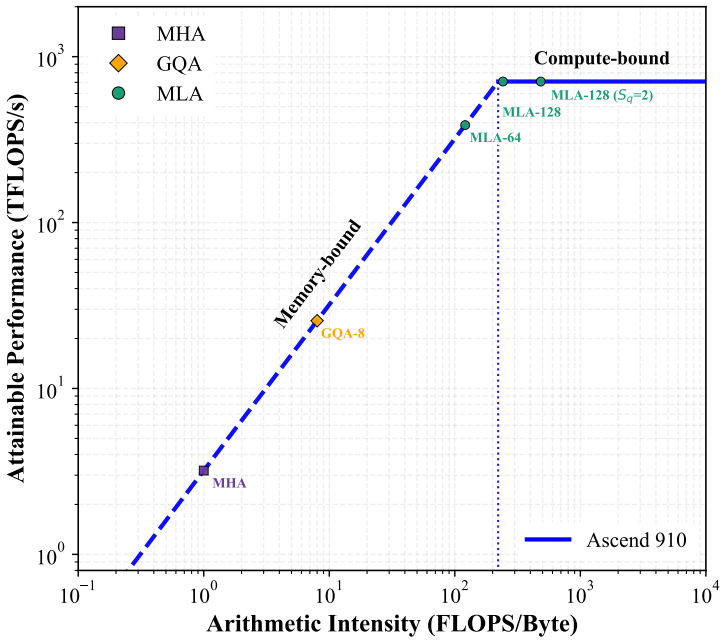

Figure 1: Roofline Analysis of BF16 Decoding on Ascend 910. The dashed segment indicates the region where performance is limited by memory bandwidth, whereas the horizontal solid lines represent the peak compute-bound performance achievable. Data points corresponding to different attention variants are plotted, showcasing their operational regimes and proximity to the hardware limits.

projecting key-value representations into a shared low-dimensional latent space, preserving model quality while drastically shrinking the KVCache.

However, this efficiency comes at a cost: MLA shifts the computational profile from memory-intensive to compute-intensive. As shown in Fig. 1, while Multi-Head Attention (MHA) and GQA remain constrained by memory bandwidth due to low arithmetic intensity (FLOPS/Byte), MLA's latent sharing significantly increases compute intensity, making it ideal for AI accelerators, especially when combined with Multi-Token Prediction (MTP), which further amplifies compute demand. This synergy makes MLA particularly well-suited for high-performance NPUs like the Ascend 910 which is designed for high FLOPS density and energy efficiency.

Yet, naive MLA implementations fail to exploit the Ascend 910's full potential. Two key bottlenecks emerge:

1. **Large Output Tensors**. Following the Flash Attention algorithm [6], the output $\mathbf{O}_i$ is rescaled frequently. However, the rescaling of output $\mathbf{O}_i \leftarrow \mathbf{O}_{i-1} \cdot \exp(m_{i-1} - m_i) + \mathbf{P}_i \mathbf{V}_i$ generates large intermediate tensors (e.g., $\mathbf{O} \in \mathbb{R}^{128 \times 512}$ in FP32). Due to the large size of the output matrix $\mathbf{O}$, these tensors cannot fit within the register file of a single Streaming Multiprocessor (SM) on GPU architectures. For the Ascend 910, $\mathbf{O}$ must be repeatedly shuttled between Global Memory (GM) and Unified Buffer (UB) — GM and UB are storage structures at different levels.
2. **Underutilized Heterogeneous Compute**. Current kernels serialize Cube (attention score matmul) and Vector (softmax, scaling) operations, leaving cores idle. The lack of pipelining between physically separated compute units results in suboptimal FLOPS utilization, which is a critical waste on hardware designed for parallelism.

Although prior work, such as FlashMLA [8], mitigates the bottleneck via reducing the row number of blocks and employing a complex scheduling algorithm, achieving 660 TFLOPS in compute-bound configuration on H800 SXM5. However, such an approach inevitably introduces additional overhead due to the repetitive movement and management of KVCache.

To address all the above bottlenecks, we propose **Ascend MLA (AMLA)**, a co-designed algorithm and kernel implementation that aligns MLA's compute patterns with the Ascend 910's architectural strengths. Our contributions are:

- **Replacing Multiplication with Addition**: We reformulate the output update using binary reinterpretation of FP32 as INT32, replacing floating-point multiplications with atomic integer additions. This enables *in-place* updates directly in GM, eliminating data movement between GM and UB for intermediate tensors.
- **Preload Pipeline & Hierarchical Tiling**: We introduce a *Preload Pipeline* that overlaps execution of Cube cores (matrix multiplication) and Vector cores (softmax, rescaling), ensuring the kernel remains Cube-bound. Besides, we design a *Hierarchical Tiling* strategy within the Cube stages, overlapping data movement with computation. This dual-layer optimization collectively minimizes pipeline bubbles and sustains near-peak compute utilization.

AMLA achieves **86.8% FLOPS utilization** on Ascend 910, surpassing FlashMLA's 66.7% on H800 SXM5 [8] while preserving numerical precision. The kernel is integrated into Huawei's CANN and will be open-sourced.

Paper Structure: Section 2 reviews attention variants and the Ascend 910 architecture. Section 3 details AMLA's algorithm and numerical analysis. Section 4 describes a hardware-aware implementation. Section 5 presents experiments. Section 6 concludes.

2 Background

2.1 Attention and FlashAttention

The Transformer architecture [20] has become the foundation of modern LLMs, including DeepSeek [9, 10, 11], Qwen [3, 21], ChatGPT [4, 17], and Gemini [12]. Its defining innovation, self-attention, enables dynamic modeling of contextual relationships across input tokens, surpassing the fixed receptive fields of CNNs and sequential dependencies of RNNs. For each attention head, query ($\mathbf{Q}$), key ($\mathbf{K}$), and value ($\mathbf{V}$) matrices are computed, and attention output is given by:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{D_k}}\right)\mathbf{V}, \tag{1}$$

where D_k denotes the head dimension of $\mathbf{Q}$ and $\mathbf{K}$.

While theoretically efficient, the naive implementation of Eq. (1) suffers from low hardware utilization due to excessive data movement between high-bandwidth memory (HBM) and compute units. To address this, FlashAttention [6, 5] introduces tiling, recomputation, and online softmax, reducing memory traffic and achieving over 5× speedup on GPUs. Algorithmic details are provided in Algorithm 1.

As context lengths and batch sizes grow during LLM inference, KVCache memory consumption becomes a critical scalability constraint. To mitigate this, architectures such as MQA [19] and GQA [2] have been proposed. MQA shares a single $\mathbf{K}/\mathbf{V}$ head across all query heads, drastically reducing KVCache size. GQA extends this idea by grouping query heads and assigning each group a dedicated $\mathbf{K}/\mathbf{V}$ head, striking a balance between memory efficiency and model expressiveness.

2.2 Multi-Head Latent Attention

MLA [9] further compresses KVCache by projecting key-value representations into a shared low-dimensional latent space. Specifically, the original $[\mathbf{K}, \mathbf{V}] \in \mathbb{R}^{S_2 \times (N_2 \times (D_k + D_v))}$ is compressed into a latent vector $c \in \mathbb{R}^{S_2 \times D_c}$, where N_2 is the head number of the original $\mathbf{K}/\mathbf{V}$, S_2 is the context length of KVCache, D_k and D_v are the head dimensions of $\mathbf{K}$ and $\mathbf{V}$, respectively. $D_c \ll N_2(D_k + D_v)$ is the dimension of the latent vector c. This compact c is cached instead of full $\mathbf{K}/\mathbf{V}$ tensors.

At each decoding step, c is up-projected to restore full-rank representations:

$$\mathbf{K} = c\mathbf{W}_k, \quad \mathbf{V} = c\mathbf{W}_v.$$

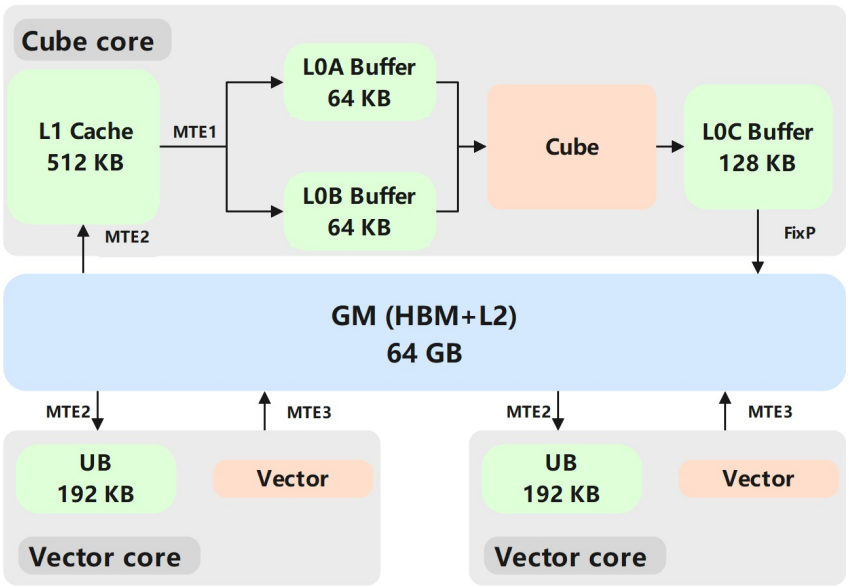

Figure 2: Da Vinci V220 architecture. Cache capacities are accessible via the Ascend C API *GetCoreMemSize*.

By precomputing $\mathbf{Q}' = \mathbf{Q}\mathbf{W}_k^T$, the attention score becomes $\mathbf{Q}'c^T$, avoiding explicit $\mathbf{K}\mathbf{K}^T$ computation. Similarly, $\mathbf{W}_v$ can be fused into the output stage. Since c is shared across all heads, MLA achieves MQA-level memory compression while preserving multi-head expressivity. Implementation details, including Rotary Position Embedding (RoPE) integration, are elaborated in [9].

Notably, MLA shifts the computational profile from memory-bound to compute-bound, which is a characteristic that aligns well with high-throughput accelerators like the Ascend 910.

2.3 Ascend 910 Architecture

The Ascend 910 NPU is a dual-die system, with each die implementing Huawei's Da Vinci V220 architecture [1] (see Fig. 2). Each die integrates 24 *Cube cores* optimized for high-throughput matrix operations and 48 *Vector cores* tailored for element-wise and reduction operations.

Hierarchical Memory Architecture. Each die features 64 GB HBM and 192 MB L2 cache, delivering 1.6 TB/s HBM bandwidth per die (3.2 TB/s aggregate). Each Cube core includes a 512 KB L1 cache and a partitioned 256 KB L0 cache (64 KB input buffers L0A/L0B, 128 KB output buffer L0C). Each Vector core utilizes a 192 KB UB as its primary working memory.

Component		Capacity	Bandwidth
HBM		64 GB	1.6 TB/s
Cube core	L1 cache	512 KB	—
	L0A cache	64 KB	—
	L0B cache	64 KB	—
	L0C cache	128 KB	—
Vector core	UB	192 KB	—

Table 1: Hierarchical memory system for each die in Ascend 910.

Cube-Vector Core Collaboration Model. The Cube and Vector cores operate with physically distinct memory spaces. Data exchange between them requires explicit movement through GM, encompassing HBM and L2 cache. This design prioritizes core specialization and scalability, enabling

concurrent execution, but requires careful orchestration in kernel design to minimize GM traffic and maximize compute overlap.

2.4 Arithmetic Intensity of Different Attentions

Arithmetic intensity is a fundamental metric in computer architecture, defined as the ratio of FLOPS to memory access (in bytes). It serves as a critical indicator for determining whether a workload is compute-bound or memory-bound. During the decode phase, tasks with high arithmetic intensity, such as MLA, are typically compute-bound, meaning their performance is constrained by the computational throughput of the accelerator. In contrast, tasks with low arithmetic intensity, such as MHA, are memory-bound, limited primarily by memory bandwidth.

The arithmetic intensity is formally expressed as:

$$\text{Arithmetic Intensity} = \frac{\text{FLOPS}}{\text{Memory Access (Bytes)}}.$$

For standard attention mechanisms, the FLOPS and memory traffic can be derived as follows:

$$\text{FLOPS} = 2N_1S_1S_2(D_k + D_v), \quad \text{MEM}_{KV} = \begin{cases} 2N_2S_2(D_k + D_v), & \text{for MHA/GQA;} \\ 2S_2D_k, & \text{for MLA.} \end{cases}$$

Here, N_1 and N_2 denote the number of query heads and key/value heads. S_1 and S_2 represent the context length of query and key/value, where S_1 is usually 1 in the decode phase, and might be a bit larger with MTP. FLOPS account for both multiplication and addition operations, and memory is measured in bytes (e.g., BF16 parameters occupy 2 bytes each).

Substituting these expressions, the arithmetic intensity becomes:

$$\text{Arithmetic Intensity} = \begin{cases} N_1S_1, & \text{for MHA/GQA;} \\ N_1S_1\frac{D_k+D_v}{D_k}, & \text{for MLA.} \end{cases}$$

Different attention mechanisms thus exhibit distinct arithmetic intensity profiles. We summarize the arithmetic intensities of MHA, GQA, and MLA under common configurations in Table 2 and visualize their characteristics in Fig. 1. For further discussion on hardware-efficient attention designs, see [23].

During the decode phase, attention mechanisms exhibit distinct arithmetic intensities (FLOPS / Byte), leading to divergent hardware utilization profiles. Different hardware-efficient variants of attention are proposed to utilize modern hardware effectively [23]. As shown in Fig. 1, MHA remains memory-bound due to low arithmetic intensity. GQA improves intensity slightly but still favors memory bandwidth. In contrast, MLA, by sharing latent representations across heads, significantly increases arithmetic intensity, making it compute-bound.

This shift renders MLA particularly well-suited for architectures with high compute density and moderate memory bandwidth — such as the Ascend 910 and NVIDIA H800. When combined with MTP, MLA's compute demand further escalates, reinforcing the need for hardware-aware optimization.

To quantify efficiency, we adopt **FLOPS Utilization (FU)** as our primary performance metric, measuring the percentage of peak theoretical compute throughput achieved during kernel execution.

2.5 FlashMLA

Proposed by DeepSeek, FlashMLA is an optimized MLA decoding kernel designed for NVIDIA Hopper GPUs [8]. To deal with the significant space occupancy of the output matrix, FlashMLA sets BLOCK_SIZE_M to be 64, which means that each FlashAttention iteration processes the computation of 64 rows. For cases where the group size exceeds 128, this approach may introduce additional data movement.

Furthermore, since the capacity of registers for each SM is 256KB, which is exactly twice $64 \times 512 \times 4 = 128$KB, it is infeasible to enable Tensor core and Cuda core at the same time if rescaling

Attention Variant	MHA	GQA	MLA-64	MLA-128	MLA-128($S_q = 2$)
Q_{head}	64	64	64	128	128
$K_{head}(V_{head})$	64	8	1	1	1
S_q	1	1	1	1	2
Arithmetic Intensity	1	8	≈ 121	≈ 242	≈ 484

Table 2: Arithmetic Intensity (FLOPS/Bytes) of Different Attentions.

of a block is performed all at once. FlashMLA splits along the column direction and implements a sophisticated "seesaw" scheduling algorithm, enabling compute-bound optimization by overlapping CUDA Core and Tensor Core operations.

Overall, by employing smaller block granularity and a complex scheduling algorithm, FlashMLA addresses the challenges of implementing the MLA kernel and achieves good performance on the H800 SXM5.

3 Algorithm

In this section, we present the AMLA algorithm and analyze its numerical accuracy and hardware efficiency.

3.1 Motivation

We begin by reviewing the original FlashAttention algorithm [6, 5] (Algorithm 1), which FlashMLA also adopts.

Algorithm 1 FlashAttention

Require: $\mathbf{Q} \in \mathbb{R}^{G\times D_k}$, $\mathbf{K} \in \mathbb{R}^{S_2\times D_k}$, $\mathbf{V} \in \mathbb{R}^{S_2\times D_v}$
Ensure: $\mathbf{O} \in \mathbb{R}^{G\times D_v}$

Initialize: $\mathbf{O} = \mathbf{0}_{G\times D_v}, m_0 = -\infty, \ell_0 = \mathbf{0}_{G\times 1}$
Partition $\mathbf{K}, \mathbf{V}$ into N blocks
for $i = 1$ to N **do**
  $\mathbf{S}_i \leftarrow \mathbf{Q}\mathbf{K}_i^T$ ▷ $[C_1]$: Compute in Cube cores
  $m_i \leftarrow \max(m_{i-1}, \text{rowmax}(\mathbf{S}_i/\sqrt{D_k}))$ ▷ $[V_1]$: Compute in Vector cores
  $\mathbf{P}_i \leftarrow \exp(\mathbf{S}_i/\sqrt{D_k} - m_i)$
  $\ell_i \leftarrow \ell_{i-1} \cdot \exp(m_{i-1} - m_i) + \text{rowsum}(\mathbf{P}_i)$
  $\mathbf{T}_i \leftarrow \mathbf{P}_i\mathbf{V}_i$ ▷ $[C_2]$: Compute in Cube cores
  $\mathbf{O}_i \leftarrow \mathbf{O}_{i-1} \cdot \exp(m_{i-1} - m_i) + \mathbf{T}_i$ ▷ $[V_2]$: Compute in Vector cores
end for
$\mathbf{O} \leftarrow \mathbf{O}_N/\ell_N$
return O

In the decode phase, inputs are $\mathbf{Q} \in \mathbb{R}^{G\times D_k}$, $\mathbf{K} \in \mathbb{R}^{S_2\times D_k}$, $\mathbf{V} \in \mathbb{R}^{S_2\times D_v}$, with output $\mathbf{O} \in \mathbb{R}^{G\times D_v}$, where typical dimensions are $G = 128$, $D_k = 576$, $D_v = 512$.

As shown in Algorithm 1, the kernel is divided into four stages: $[C_1], [V_1], [C_2], [V_2]$, where $[C_i]$ runs on Cube cores and $[V_i]$ on Vector cores.

Compared to GQA, the $[V_2]$ stage in MLA incurs significant data movement between GM and UB due to the large intermediate output matrix $\mathbf{O}_i$ ($D_v = 512$ vs. 128 in GQA). Since $\mathbf{O}_i$ is typically stored in FP32, its memory footprint is:

$$G \times D_v \times 4 \text{ Bytes} = 256 \text{ KB}$$

Due to the 1:2 ratio between Cube and Vector cores, the memory budget of residing $\mathbf{O}_i$ per Vector core will be 128 KB – already insufficient, as the 192 KB UB space must be shared with other intermediate operands. Attempting to keep $\mathbf{O}_i$ resident in UB thus introduces severe scheduling complexity and resource contention. With MTP, its memory footprint doubles, making UB residency completely infeasible. Consequently, each update in

$$\text{Update}(\mathbf{O}): \quad \mathbf{O}_i \leftarrow \mathbf{O}_{i-1} \cdot \exp(m_{i-1} - m_i) + \mathbf{P}_i \mathbf{V}_i. \tag{2}$$

requires:

1. Loading $\mathbf{O}_{i-1}, \mathbf{T}_i = \mathbf{P}_i \mathbf{V}_i$ from GM to UB, and computing Eq. (2);
2. Writing the updated $\mathbf{O}_i$ back from UB to GM.

This double memory movement introduces latency and pipeline bubbles, making $[V_2]$ the critical bottleneck for MLA on Ascend NPUs.

Naive Optimization and Its Pitfall

To eliminate data movement, one might consider transforming Eq. (2) into:

$$\widehat{\mathbf{O}}_i \leftarrow \widehat{\mathbf{O}}_{i-1} + \exp(m_i)\mathbf{P}_i \mathbf{V}_i, \quad \text{where} \quad \widehat{\mathbf{O}}_i = \exp(m_i)\mathbf{O}_i. \tag{3}$$

This allows direct in-GM updates via **AtomicAdd**. However, $\exp(m_i)$ may exceed the representable range of FP32, leading to overflow. For example, when $m_i > 88$, $\exp(m_i)$ overflows in FP32. Actually, Eq. (3) is the naive softmax without safe processing, and this is why safe softmax is widely used [15]. Thus, this naive approach fails numerically.

3.2 AMLA: Stable In-Memory Updates

We propose a numerically stable transformation:

$$\tilde{\mathbf{O}}_i \leftarrow \tilde{\mathbf{O}}_{i-1} \cdot 2^{n_i - n_{i-1}} + \frac{1}{r_i}\mathbf{P}_i \mathbf{V}_i, \tag{4}$$

where

$$\tilde{\mathbf{O}}_i = \mathbf{O}_i / r_i, \quad n_i = \text{round}(-m_i / \ln 2), \quad r_i = \exp(-n_i \ln 2 - m_i).$$

Here, $1/\sqrt{2} \le r_i \le \sqrt{2}$, ensuring numerical stability. The term $\frac{1}{r_i}\mathbf{P}_i \mathbf{V}_i$ can be fused into the $[V_1]$ stage without extra memory access.

The key innovation lies in computing $\tilde{\mathbf{O}}_{i-1} \cdot 2^{n_i - n_{i-1}}$ *in-place* in GM using **AtomicAdd**, leveraging the IEEE 754 floating-point representation.

Multiplication via Integer Addition

Let's first review the FP32 format. The IEEE 754 single-precision (FP32) format encodes a 32-bit value as three fields [13]: a sign bit S (bit 31), 8 exponent bits E (bits 23–30), and 23 mantissa bits M (bits 0–22). The corresponding floating-point value, denoted as F, is given by:

$$F = (-1)^S \times \left(1 + \frac{M}{2^{23}}\right) \times 2^{E-127}, \quad S \in \{0, 1\},\ 0 < E < 255,\ 0 \le M < 2^{23}. \tag{5}$$

We neglect subnormal values and do not consider INF/NAN hereafter, as they are irrelevant in typical LLM attention computations.

Simultaneously, interpreting the same 32-bit pattern as a signed integer (INT32), denoted as I, yields:

$$I = -2^{31}S + 2^{23}E + M. \tag{6}$$

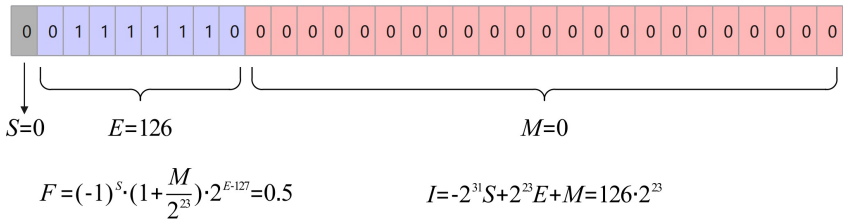

Figure 3: The bit pattern 00111111000000000000000000000000 is 0.5 when interpreted as an FP32 value, and 126×2^{23} when interpreted as an INT32 number.

Example 3.1. *The 32-bit pattern 00111111000000000000000000000000 represents the value 0.5 when interpreted as an FP32 number, and represents* 126×2^{23} *when interpreted as an INT32 number, see Fig. 3*

This reinterpretation enables a critical insight: multiplying a normalized FP32 value by a power of two (2^k) corresponds to adding k to its exponent field E, which is a simple fixed-offset addition when reinterpreted as an integer. This allows us to perform exponential rescaling *in-place* in GM via **AtomicAdd**, without loading the full tensor into compute units.

Defining **AS_INT32** and **AS_FP32** as bit-preserving reinterpretations, we have

$$\text{AS\_INT32}(F) = I, \quad \text{AS\_FP32}(I) = F. \tag{7}$$

Given the above definition, we establish the following lemma:

Lemma 3.1. *Given an FP32 number F, let $I = AS\_INT32(F)$ denote the integer represented by its binary pattern. Suppose $0 < E < 255$ is the unsigned integer value represented by F's exponent bits. Then, for any integer $n \in \mathbb{Z}$ satisfying $-E < n < 255 - E$, the result of the multiplication $F \times 2^n$ shares the same bit pattern as the result of the integer addition $I + n \times 2^{23}$.*

Proof. Let S, E, and M denote the numerical values represented by the sign bit (bit 31), exponent bits (bits 23–30), and mantissa bits (bits 0–22) of F, respectively. Then F is expressed as Eq. (5). Let I in Eq. (6) be the integer corresponding to F's binary representation.

Since $0 < E + n < 255$ remains within the representable range of UINT8,

$$F' = F \times 2^n = (-1)^S \times (1 + \frac{M}{2^{23}}) \times 2^{(E+n)-127},$$

and the INT32 integer corresponding to its binary representation is

$$I' = -2^{31} \times S + 2^{23} \times (E + n) + M = I + n \times 2^{23}.$$

Evidently, $F' = F \times 2^n$ and $I' = I + n \times 2^{23}$ share identical underlying binary representations. Therefore, $F \times 2^n$ can be implemented via a single integer addition $I + n \times 2^{23}$, that is

$$F \times 2^n = \text{AS\_FP32}(\text{AS\_INT32}(F) + n \times 2^{23}). \tag{8}$$

□

Remark 3.1. *This paper presents results for FP32, as it remains the most prevalent precision in large language model training; however, the algorithm is also applicable to other data types like TF32, BF16, and even FP64.*

Remark 3.2. *In Algorithm 2, the row scaling operation on Line 10 can be directly integrated into Line 7 and implemented via vector subtraction to reduce a portion of the computational load.*

Algorithm 2 Ascend MLA (AMLA)

Require: $\mathbf{Q} \in \mathbb{R}^{G \times D_k}$, $\mathbf{K} \in \mathbb{R}^{S_2 \times D_k}$, $\mathbf{V} \in \mathbb{R}^{S_2 \times D_v}$
Ensure: $\mathbf{O} \in \mathbb{R}^{G \times D_v}$

1: Initialize: $\mathbf{O} = \mathbf{0}_{G \times D_v}, m_0 = -\infty, \ell_0 = \mathbf{0}_{G \times 1}, n_0 = -\infty, c_0 = 1$
2: Partition $\mathbf{K}, \mathbf{V}$ into N blocks
3: **for** $i = 1$ to N **do**
4: $\quad \mathbf{S}_i \leftarrow \mathbf{Q}\mathbf{K}_i^T$
5: $\quad \mathbf{S}_i \leftarrow \mathbf{S}_i / \sqrt{D_k}$, $m_i \leftarrow \max(m_{i-1}, \text{rowmax}(\mathbf{S}_i))$, $M_{\text{up}} \leftarrow \exp(m_{i-1} - m_i)$
6: $\quad n_i \leftarrow \text{round}(-m_i / \ln 2)$, $\mathbf{P}_i \leftarrow \exp(\mathbf{S}_i - m_i)$, $\ell_i \leftarrow \ell_{i-1} \cdot M_{up} + \text{rowsum}(\mathbf{P}_i)$
7: $\quad S_{32} \leftarrow \exp(\ln 2 \cdot (n_i + m_i / \ln 2))$
8: $\quad S_{16} \leftarrow$ Cast_to_BF16(S_{32}), then back to FP32
9: $\quad c_i \leftarrow S_{32} / S_{16}$, $\varepsilon \leftarrow 1.5 \cdot (c_i / c_{i-1} - 1)$
10: $\quad \mathbf{P}_i \leftarrow \mathbf{P}_i \cdot S_{16}$, then cast to BF16
11: $\quad N \leftarrow \max(n_i - n_{i-1}, -30) + \varepsilon + 10^{-6}$
12: $\quad N \leftarrow$ Cast_to_int32($N \cdot 2^{23}$)
13: $\quad$ **if** $i > 1$ **then**
14: $\quad\quad \mathbf{O} \leftarrow$ AS_INT32($\mathbf{O}$) $+ N$ ▷ **AtomicAdd⟨INT32⟩ in GM**
15: $\quad\quad \mathbf{O} \leftarrow$ AS_FP32($\mathbf{O}$)
16: $\quad$ **end if**
17: $\quad \mathbf{O}_i \leftarrow \mathbf{P}_i \mathbf{V}_i$
18: $\quad \mathbf{O} \leftarrow \mathbf{O} + \mathbf{O}_i$ ▷ **AtomicAdd⟨FP32⟩ in GM**
19: **end for**
20: $\mathbf{O} \leftarrow \mathbf{O} / (\ell_N \cdot S_{16})$
21: **return** $\mathbf{O}$

In-Memory Update with AtomicAdd

Applying Lemma 3.1, we compute:

$$\tilde{\mathbf{O}}_{i-1} \cdot 2^{n_i - n_{i-1}} = \text{AS\_FP32}\left(\text{AS\_INT32}(\tilde{\mathbf{O}}_{i-1}) + (n_i - n_{i-1}) \cdot 2^{23}\right). \tag{9}$$

This enables in-GM updates via **AtomicAdd**. Combined with accumulating $\frac{1}{r_i}\mathbf{P}_i\mathbf{V}_i$, the entire $\tilde{\mathbf{O}}_i$ update occurs in memory — eliminating transfers between GM and UB.

Error Compensation

A subtle precision issue arises due to BF16 quantization in intermediate steps; we address it via error compensation (see Section A).

3.3 Comparison between Base and AMLA

We recall **Base** as the standard FlashAttention-style implementation (see Algorithm 1), which decomposes attention into four distinct pipeline stages: $[C_1]$ ($\mathbf{QK}^T$ on Cube cores), $[V_1]$ (online softmax on Vector cores), $[C_2]$ ($\mathbf{PV}$ on Cube cores), and $[V_2]$ (rescaling on Vector cores). In MLA, the $[V_2]$ stage becomes a severe bottleneck due to the large FP32 output tensor $\mathbf{O}_i$, forcing repeated transfers between GM and UB that stall the pipeline and underutilize compute units.

AMLA eliminates this bottleneck by collapsing $[V_2]$ via numerically stable in-memory computations: Rescaling is converted to integer addition (via **AtomicAdd⟨INT32⟩**), and output accumulation occurs directly in GM (via **AtomicAdd⟨FP32⟩**). FlashAttention-2 conducts the normalization in the final Vector stage [5], which is also adopted by AMLA (Line 20 in Algorithm 2). Apart from the final one, the pipeline is reduced to three stages ($[C_1], [V_1], [C_2]$) without sacrificing numerical accuracy (validated in Section 5.1), while the repeated data movement in $[V_2]$ is eliminated at the same time. The comparison between Base and AMLA in the rescaling phase is shown in Fig. 4.

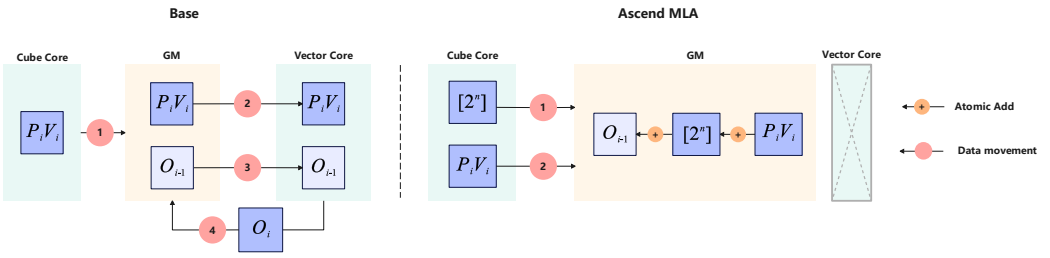

Figure 4: Base vs. Ascend MLA.

4 Implementation

This section presents hardware-aware optimizations tailored for Ascend NPUs. We achieve two levels of pipelining: (1) *inter-stage* overlap between Cube and Vector cores across $[C_1]$, $[V_1]$, $[C_2]$; and (2) *intra-stage* overlap of computation and memory movement within each Cube stage. As introduced in Section 3, AMLA consists of three core stages: $[C_1]$ and $[C_2]$ (matrix multiplication on Cube cores), and $[V_1]$ (vector operations on Vector cores).

- Section 4.1 introduces the **Preload Pipeline**, which decouples inter-stage dependencies, enabling full overlap of Vector stages by concurrent Cube stages. This renders the kernel **Cube-bound** — its performance is ultimately limited by Cube core latency.
- As established in Section 2.4, MLA is inherently compute-bound. Thus, within Cube stages, we further optimize for **computation-memory overlap**. Section 4.2 details our hierarchical tiling and pipeline design, derived from theoretical bandwidth-compute balance analysis, to maximize Flops utilization.

4.1 Preload Pipeline

This subsection introduces the Preload Pipeline, a novel two-phase architecture designed to eliminate inter-stage dependencies and achieve full hardware utilization. Our presentation is structured as follows: first, we outline the core architectural design, which separates an initial dependency-resolving *Preload* phase from a stall-free *Steady Pipeline Loop*. Second, we formalize its efficiency by defining the *Preload count*, a metric for warm-up overhead, and derive a tight theoretical bound for its minimization. Finally, we demonstrate the pipeline's practical application to the AMLA kernel. To ground this discussion, we begin with the generalized dependency chain common in such workloads, $[C_1] \to [V_1] \to [C_2] \to [V_2]$ (where AMLA is a special case with $[V_2] = 0$), illustrating how naive execution leads to underutilization.

4.1.1 Two-phase Pipeline Overview

We first propose a two-phase pipeline architecture: *Preload* and *Steady Pipeline Loop* (see Fig. 5).

1. **Preload:** Resolves upstream dependencies before the main loop begins, enabling flexible scheduling of the first cycle (Cycle 1 in Fig. 5). For instance, if $[C_2]$ is scheduled before $[C_1]$ (despite $[C_2]$ depending on $[V_1]$), the Preload phase pre-executes $[C_1]$ and $[V_1]$ to satisfy $[C_2]$'s dependencies ($[C_1] \to [V_1] \to [C_2]$).
2. **Steady Pipeline Loop:** Repeated execution of identical *Cycles*, each encapsulating $[C_1]$, $[V_1]$, $[C_2]$, $[V_2]$. Thanks to dependency decoupling in Preload, operations within a Cycle can be reordered freely without stalls.

This architecture generalizes to arbitrary n CV-pair chains and imposes no constraints on individual stage durations or aggregate Cube/Vector timing, ensuring optimal performance regardless of whether the workload is Cube-bound or Vector-bound.

4.1.2 Key Conclusions

We define the *Preload count* as the minimum number of $[C_1]$ stages executed in the Preload phase. This metric is equivalent to the number of operation blocks per Cycle that require external dependency

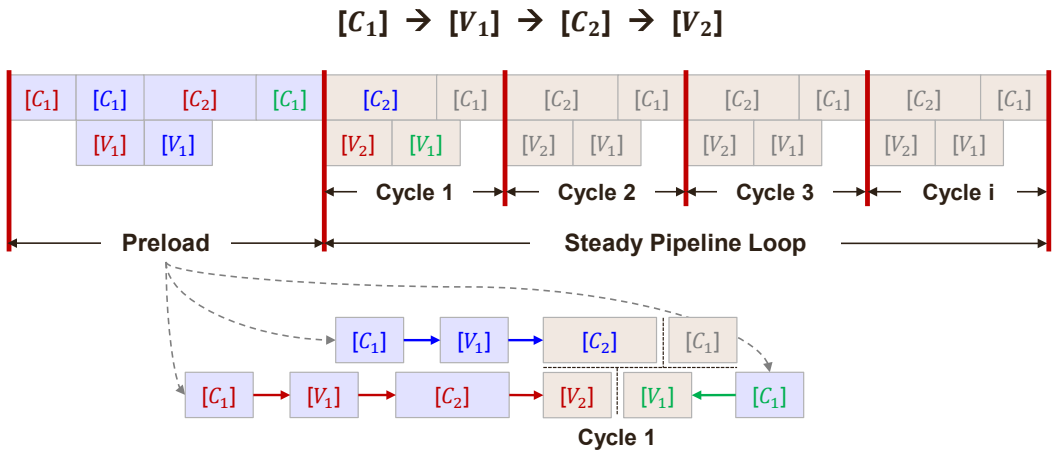

Figure 5: Two-phase pipeline: (1) Preload resolves initial dependencies; (2) Steady Loop executes Cycles with maximal concurrency.

resolution. For example, the Preload number sin Fig. 5 is 3, since there are 3 $[C_1]$ stages in Preload. As illustrated in Fig. 6, the Preload count can be minimized by reordering intra-Cycle blocks, which directly reduces the pipeline warm-up duration.

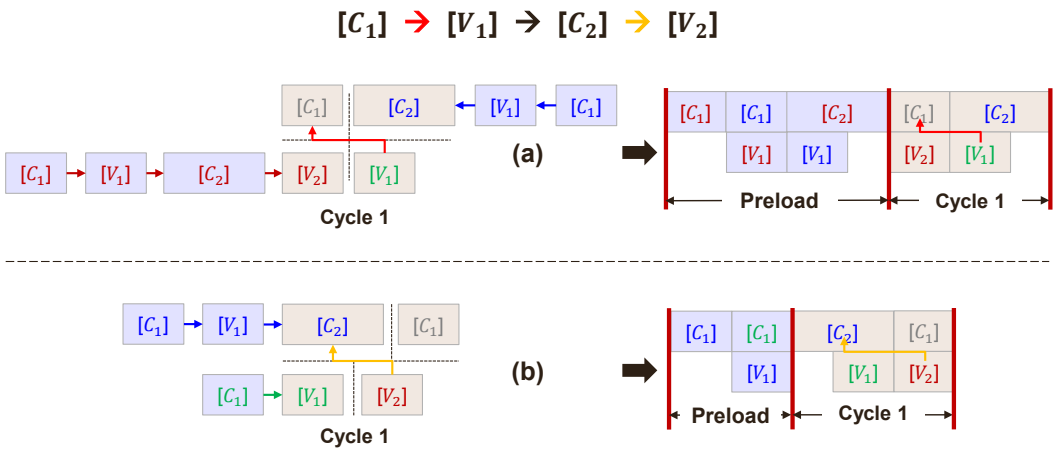

Figure 6: Rearrange Cycles to shorten the Preload phase. (a) Decoupling an earlier dependency $[C_1] \rightarrow [V_1]$ reduces the Preload phase by one operation block ($[C_1]$) compared with Fig. 5. (b) Decoupling a latter dependency $[C_2] \rightarrow [V_2]$ eliminates three operation blocks ($[C_1]$, $[V_1]$, $[C_2]$) from the Preload phase compared with (a), highlighting that rearranging the internal operation blocks of a Cycle achieves a more significant reduction in Preload.

Combining the upper bound (Lemma B.2) and existence guarantee (Theorem B.1) from Appendix B, we derive a tight theoretical guarantee:

Theorem 4.1. *For any n-stage $[C][V]$ chain $[C_1] \rightarrow [V_1] \rightarrow \cdots \rightarrow [C_n] \rightarrow [V_n]$, where the latency of each $[C_i]$ and $[V_i]$ can be arbitrary:*

1. *A pipeline with Preload count $< n$ is not always achievable.*
2. *A pipeline with Preload count $= n$ always exists, with all intra-Cycle dependencies of the form $[V] \rightarrow [C]$.*

This bound ensures minimal pipeline warm-up and maximal hardware utilization: the Preload phase resolves initial dependencies efficiently, while the Steady Loop sustains stall-free execution, eliminating idle cycles and achieving near-peak Cube core utilization.

4.1.3 Preload Pipeline for AMLA

The above principles of the Preload pipeline are directly applied in the AMLA kernel, as illustrated in Fig. 7. AMLA can be viewed as a specific instance of the above general model where the number of CV pairs $n = 2$ (with stages $[C_1] \rightarrow [V_1] \rightarrow [C_2]$) and the final vector stage $[V_2]$ has a duration of zero. In accordance with Theorem 4.1, we adopt a Preload count of 2: in the Preload phase, we first execute $[C_1]$ and $[V_1]$ to decouple $[C_2]$ of the first Cycle, and then execute $[C_1]$ to decouple $[V_1]$ of the first Cycle. After entering the Steady Pipeline Loop, each Cycle encapsulates $[[C_1], [C_2], [V_1]]$ with no intra-Cycle dependencies. At the same time, in Steady Pipeline Loop, the Vector stage is fully overlapped by the ongoing Cube stage at the outer level, and the kernel is Cube-bound. Since the Preload phase issues $[[C_1], [C_1], [V_1]]$, we drain the pipeline at the tail by appending the remaining downstream stages $[[C_2], [C_2], [V_1]]$ and a Last $[V]$, thereby closing the dependency chain and completing the operator.

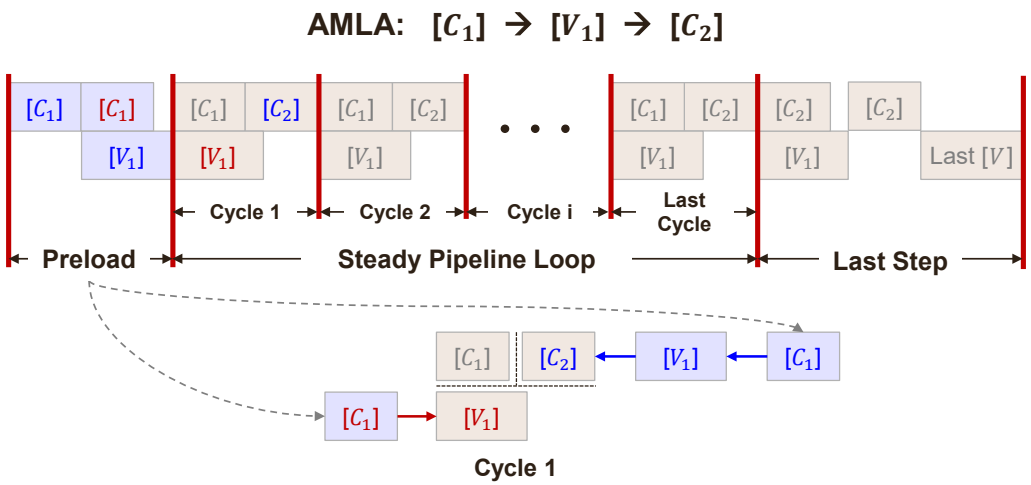

Figure 7: Preload pipeline architecture for AMLA

4.2 Hierarchical Tiling

In this subsection, we detail our optimized tiling and pipeline strategy. We categorize the Cube stages into four distinct pipelines:

- **MMAD**. Multiply–accumulate operations.
- **MTE1**. Data movement from L1 cache to L0A/L0B cache.
- **MTE2**. Data movement from GM to L1 cache.
- **FixP**. Data movement from L0C cache to GM.

Both $[C_1]$ and $[C_2]$ follow the pipeline order:

$$\textbf{MTE2} \rightarrow \textbf{MTE1} \rightarrow \textbf{MMAD} \rightarrow \textbf{FixP}. \tag{10}$$

We design a **hierarchical tiling strategy** that balances **MTE2/MTE1/FixP** bandwidth and Cube core compute throughput. Let M, N, K denote the tensor dimensions per FlashAttention iteration. We fix the KV block size to 512, yielding:

- $[C_1]$: $N = 512, K = 576$
- $[C_2]$: $N = K = 512$

Data is tiled hierarchically:

- From GM to L1: tiled as $singleM \times singleK$, $singleN \times singleK$
- From L1 to L0A/B: further tiled as $baseM \times baseK$, $baseN \times baseK$ (see Fig. 8)

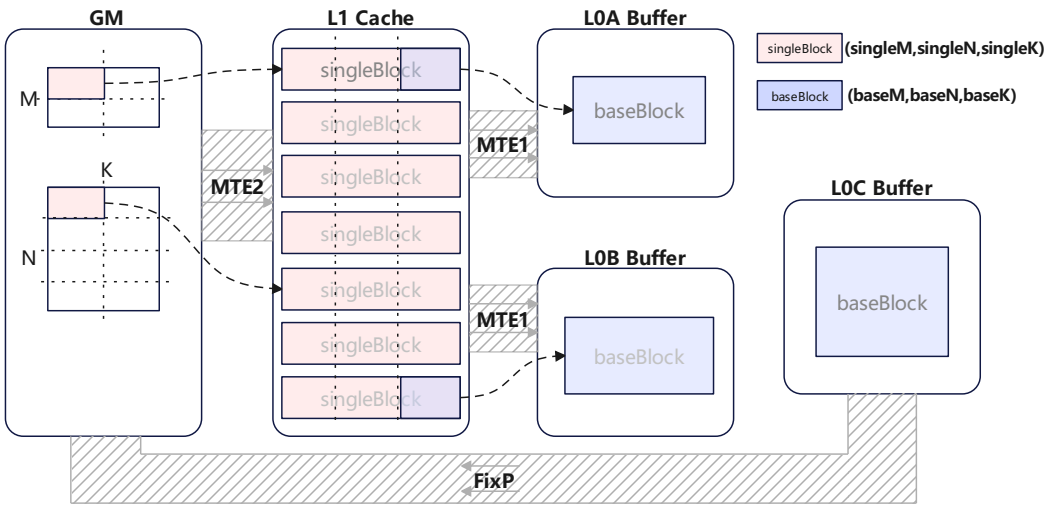

Figure 8: Hierarchical tiling across memory hierarchy.

FlashAttention Block Size. To overlap data transfer with computation, we require:

$$\frac{M \cdot N \cdot K \cdot 2}{n_{\mathrm{c}} \cdot f \cdot F_{\mathrm{BF16}}} \geq \frac{N \cdot K \cdot \mathrm{sizeof(BF16)}}{\mathrm{BW_{HBM}}}.$$

In the case of Ascend 910, the number of Cube cores is $n_{\mathrm{c}} = 48$, HBM bandwidth is $\mathrm{BW_{HBM}} = 3.2$ TB/s, f is the frequency, and F_{BF16} is the peak BF16 FLOPS for each Cube core per clock cycle.

We exclude the **MTE2** overhead of $\mathbf{Q}$ and $\mathbf{P}$ because: (1) they reside in L1 within an iteration; (2) after initial load, subsequent accesses are served from L2 Cache. For Ascend 910, we set $M = 256$ (DeepSeek-V3, 128 query heads) to balance compute and bandwidth.

L1 Cache Tiling. We retain $\mathbf{Q}$/$\mathbf{P}$ in L1 within each FlashAttention loop to minimize redundant KVCache transfers. We reserve 288 KB for $\mathbf{Q}$ (also used for $\mathbf{P}$). To avoid the long warm-up overhead of memory access, we pipeline transfers in 128×288 ($\mathbf{Q}$) or 128×256 ($\mathbf{P}$) stripes.

The remaining 224 KB L1 space holds $\mathbf{K}$ (512×576) and $\mathbf{V}$ (512×512) via a 3-buffer pipeline (72 KB/buffer):

- $[C_1]$: $singleM = 128$, $singleK = 288$, $singleN = 256$
- $[C_2]$: $singleM = 128$, $singleK = 256$, $singleN = 256$

The 512 KB L1 cache is partitioned into seven 72 KB buffers: four for $\mathbf{Q}$/$\mathbf{P}$, three for $\mathbf{K}$/$\mathbf{V}$ (see Fig. 8).

Remark 4.1. *$[C_1]$ and $[C_2]$ share identical L1 Cache tiling to eliminate pipeline bubbles between stages.*

L0 Cache Tiling. With double buffering and L0 sizes (L0A/B: 64 KB, L0C: 128 KB), we enforce:

$$\begin{aligned} baseM \cdot baseK \cdot \mathrm{sizeof(BF16)} &\leq 32\,\mathrm{KB}, \\ baseN \cdot baseK \cdot \mathrm{sizeof(BF16)} &\leq 32\,\mathrm{KB}, \\ baseM \cdot baseN \cdot \mathrm{sizeof(FP32)} &\leq 64\,\mathrm{KB}. \end{aligned}$$

To fully exploit L0 cache, we set $baseM = baseN = 128$, and adjust $baseK$ per stage:

- $[C_1]$: $baseK = 96$ (to match 576 input dim)
- $[C_2]$: $baseK = 128$ (to match 512 input dim)

This ensures balanced compute load per **MMAD**.

FixP. To minimize writeback traffic, we accumulate multiple results in L0C before bulk transfer to GM.

The resulting pipeline (see Fig. 9) achieves perfect overlap of memory and compute, ensuring the AMLA kernel remains strictly compute-bound.

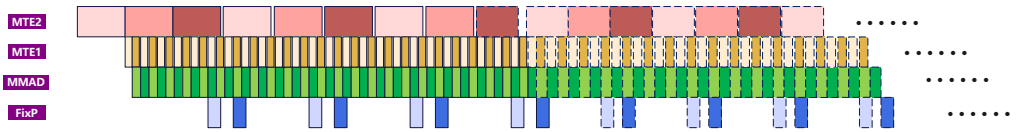

Figure 9: Optimized Cube stage pipeline. Colors denote buffer partitions (triple-buffer L1 for $\mathbf{K}$ and $\mathbf{V}$, double-buffer L0). Solid/dashed lines: $[C_1]$/$[C_2]$ stages. The MTE2 of $\mathbf{Q}$ and $\mathbf{P}$ is not included.

5 Numerical Results

In this section, we evaluate the accuracy and performance of AMLA through comprehensive experiments.

5.1 Accuracy

We evaluate AMLA's accuracy by comparing it against two baselines: **Golden** and **Base**:

- **Golden**: High-precision FP32 attention computation implemented on CPU, serving as the ground-truth reference.
- **Base**: CPU simulation of the standard FlashAttention implementation using mixed-precision matrix multiplications, with final outputs in FP16 or BF16.
- **AMLA**: NPU implementation based on Algorithm 2.

To quantify accuracy, we compute the relative Frobenius-norm error between each method's output and the Golden reference:

$$\mathcal{E}(A, B) = \frac{\|A - B\|_F}{\|B\|_F + \varepsilon},$$

where $\|C\|_F = \sqrt{\mathrm{Tr}(CC^H)} = \sqrt{\sum_{i,j} |c_{ij}|^2}$, and $\varepsilon = 10^{-10}$ for numerical stability.

We test across multiple input distributions for $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ in BF16: Gaussian $\mathcal{N}(0, \sigma^2)$ with varying σ, and uniform $\mathcal{U}(a, b)$ with different ranges. For each distribution, we generate 100 random samples under typical settings (context length = 8K) and report average errors. As shown in Table 3 and Table 4, AMLA achieves accuracy nearly identical to Base.

$\mathcal{E}(\cdot, \text{Golden})$	$\mathcal{N}(0,1)$	$\mathcal{N}(0,4)$	$\mathcal{N}(0,9)$	$\mathcal{N}(0,16)$	$\mathcal{N}(0,25)$	$\mathcal{N}(0,100)$
Base	1.77E-03	1.74E-03	1.65E-03	1.51E-03	1.33E-03	7.82E-04
AMLA	1.81E-03	1.75E-03	1.66E-03	1.51E-03	1.35E-03	7.86E-04

Table 3: Accuracy comparison under Gaussian input distributions.

$\mathcal{E}(\cdot, \text{Golden})$	$\mathcal{U}(-1,1)$	$\mathcal{U}(-3,3)$	$\mathcal{U}(-5,5)$	$\mathcal{U}(-10,10)$	$\mathcal{U}(-20,20)$	$\mathcal{U}(-60,60)$
Base	1.97E-03	1.77E-03	1.69E-03	1.24E-03	7.04E-04	2.26E-04
AMLA	2.01E-03	1.78E-03	1.69E-03	1.24E-03	7.04E-04	2.26E-04

Table 4: Accuracy comparison under uniform input distributions.

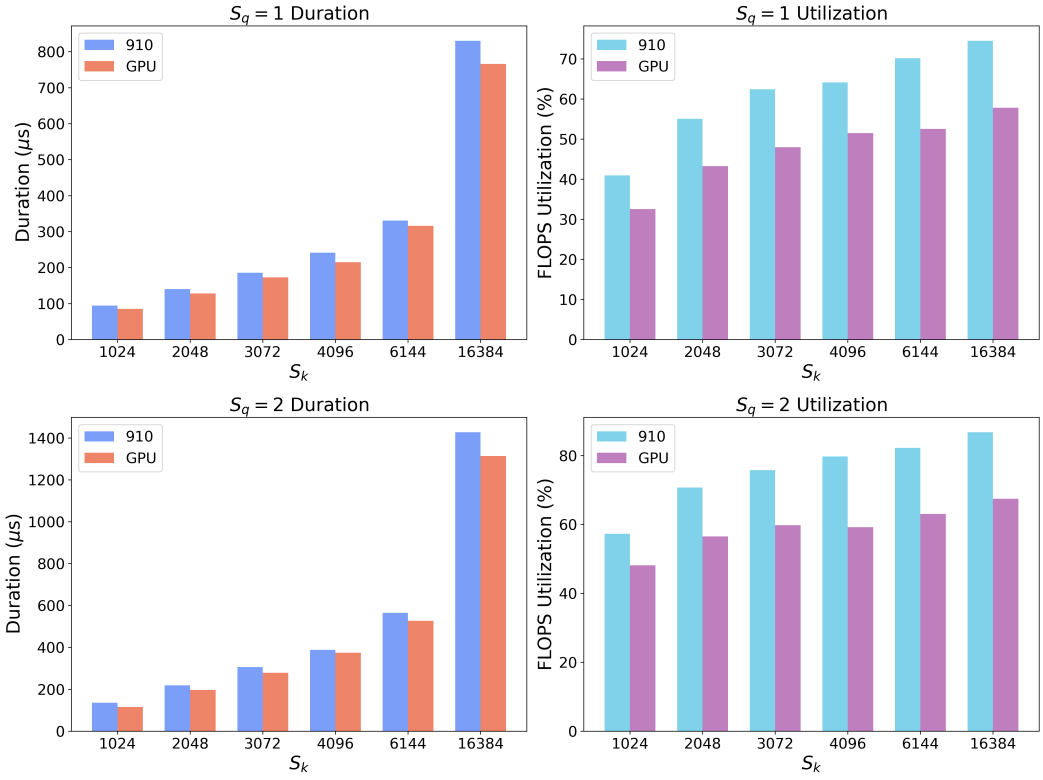

Figure 10: Performance comparison.

5.2 Performance

As a core component in LLM inference, the MLA kernel's latency significantly impacts overall system performance, especially in long-context scenarios. With the adoption of DeepEP [7] and prefill-decode disaggregation [24], the computational burden of MoE layers is reduced, making attention kernel efficiency a key bottleneck for throughput.

Leveraging our algorithmic and hardware-aware optimizations, we implement the AMLA kernel on Ascend 910 to maximize hardware utilization.

We benchmark AMLA on Ascend 910 against a GPU with 989 TFLOPS of BF16 compute performance and 3.35 TB/s memory bandwidth in the decode phase. Experiments use typical BF16 configurations. Since tensor parallelism is typically not applied to the MLA component, we set the number of query heads to 128 and key heads to 1. Batch size is fixed at 96. We evaluate both non-MTP ($S_q = 1$) and MTP ($S_q = 2$) scenarios, with variable context lengths S_k.

The comparison of kernel duration and FLOPS utilization is shown in Fig. 10, and the specific results are shown in Table 5. Based on the experimental results, AMLA achieves up to **86.8%** FLOPS utilization, a testament to our hardware-aware optimizations.

6 Summary

In this work, we present AMLA — a highly optimized MLA kernel tailored for efficient decoding on Ascend 910 NPUs. Our key contribution lies in a synergistic combination of algorithmic innovation and hardware-aware optimization, enabling unprecedented efficiency in attention computation for long-context LLM inference.

First, by exploiting the binary correspondence between floating-point and integer representations, we reformulate the rescaling of $\mathbf{O}$ blocks to replace multiplications with additions. This eliminates redundant arithmetic operations and reduces associated memory traffic. Second, we introduce

(a) Part 1

	S_k	1024		2048		3072	
S_q	Hardware	duration (μs)	FU	duration (μs)	FU	duration (μs)	FU
1	910	95	40.9%	140	55.1%	186	62.4%
	GPU	85	32.6%	128	43.3%	173	48.0%
2	910	135	57.3%	219	70.7%	306	75.8%
	GPU	115	48.1%	196	56.5%	278	59.8%

(b) Part 2

	S_k	4096		6144		16384	
S_q	Hardware	duration (μs)	FU	duration (μs)	FU	duration (μs)	FU
1	910	241	64.1%	331	70.2%	830	74.5%
	GPU	215	51.5%	316	52.6%	766	57.8%
2	910	388	79.7%	565	82.2%	1427	**86.8%**
	GPU	374	59.2%	527	63.0%	1314	67.4%

Table 5: Performance comparison. FU denotes FLOPS utilization.

hardware-aware optimizations, including a Preload Pipeline strategy and optimized hierarchical tiling, which fully leverage Ascend NPU's native support for in-memory computing and fine-grained orchestration of heterogeneous compute units.

Evaluated on Ascend 910, AMLA achieves up to **86.8%** FLOPS utilization — surpassing the state-of-the-art FlashMLA [8] (67.4% utilization) by a significant margin.

Together with our accuracy validation, this work provides a complete, high-performance, and production-ready solution for deploying MLA-based attention on Ascend NPUs.

References

[1] NPU architecture version 220. `https://www.hiascend.com/document/detail/zh/CANNCommunityEdition/83RC1alpha002/opdevg/Ascendcopdevg/atlas_ascendc_10_0011.html`.

[2] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, et al. GQA: Training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Stroudsburg, PA, USA, 2023. Association for Computational Linguistics.

[3] Jinze Bai, Shuai Bai, Yunfei Chu, et al. Qwen Technical Report. *arXiv preprint arXiv:2309.16609*, 2023.

[4] Tom B. Brown, Benjamin Mann, Nick Ryder, et al. Language Models are Few-Shot Learners. *arXiv preprint arXiv:2005.14165*, 2020.

[5] Tri Dao. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. *International Conference on Learning Representations*, 2024.

[6] Tri Dao, Dan Fu, Stefano Ermon, et al. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

[7] DeepSeek. DeepEP, 2025. `https://github.com/deepseek-ai/DeepEP`.

[8] DeepSeek. FlashMLA, 2025. `https://github.com/deepseek-ai/FlashMLA`.

[9] DeepSeek-AI. DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model, 2024. https://arxiv.org/abs/2405.04434.

[10] DeepSeek-AI. DeepSeek-V3 Technical Report. *arXiv preprint arXiv:2412.19437*, 2024.

[11] DeepSeek-AI. Deepseek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. *arXiv preprint arXiv:2501.12948*, 2025.

[12] Google DeepMind Gemini Team. Gemini: A Family of Highly Capable Multimodal Models. *arXiv preprint arXiv:2312.11805*, 2023.

[13] IEEE. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.

[14] Yiwei Li, Huaqin Zhao, Hanqi Jiang, et al. Large Language Models for Manufacturing. *arXiv preprint arXiv:2410.21418*, 2024.

[15] Maxim Milakov and Natalia Gimelshein. Online normalizer calculation for softmax. *arXiv preprint arXiv:1805.02867*, 2018.

[16] Yuqi Nie, Yaxuan Kong, Xiaowen Dong, et al. A Survey of Large Language Models for Financial Applications: Progress, Prospects and Challenges. *arXiv preprint arXiv:2406.11903*, 2024.

[17] OpenAI. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774*, 2023.

[18] Libo Qin, Qiguang Chen, Xiachong Feng, et al. Large Language Models Meet NLP: A Survey. *arXiv preprint arXiv:2405.12819*, 2024.

[19] Noam Shazeer. Fast Transformer Decoding: One Write-Head is All You Need. *arXiv preprint arXiv:1911.02150*, 2019.

[20] Ashish Vaswani, Noam Shazeer, Niki Parmar, et al. Attention Is All You Need. *Advances in neural information processing systems*, 30, 2017.

[21] Chenfei Wu, Jiahao Li, Jingren Zhou, et al. Qwen-image technical report, 2025.

[22] Shunyu Yao, Jeffrey Zhao, Dian Yu, et al. REACT: Synergizing reasoning and acting in language models. *International Conference on Learning Representations*, 2023.

[23] Ted Zadouri, Hubert Strauss, and Tri Dao. Hardware-Efficient Attention for Fast Decoding. *arXiv preprint arXiv:2505.21487*, 2025.

[24] Pengfei Zuo, Huimin Lin, Junbo Deng, et al. Serving Large Language Models on Huawei CloudMatrix384. *arXiv preprint arXiv:2506.12708*, 2025.

A Error Compensation in AMLA

Although Eq. (4) is mathematically equivalent to Eq. (2), its numerical behavior under finite-precision floating-point arithmetic differs significantly in practice. While FP32 computation introduces negligible error, especially since the final output is typically cast to BF16, the use of low-precision matrix multiplications (e.g., $\mathbf{P}_i\mathbf{V}_i$) on AI accelerators introduces non-negligible rounding errors that accumulate across iterations.

Specifically, the on-chip computation of Eq. (4) can be expressed as:

$$\tilde{\mathbf{O}}_i = \tilde{\mathbf{O}}_{i-1} \times 2^{n_i - n_{i-1}} + \frac{1}{r_i} \left(\mathbf{P}_i\right)_{\text{BF16}} \left(\mathbf{V}_i\right)_{\text{BF16}}, \tag{11}$$

where $(\cdot)_{\text{BF16}}$ denotes explicit casting to BF16 to reduce both computational cost and memory traffic. Modern specialized hardware units, including Ascend Cube cores, typically accept BF16 inputs and perform internal accumulation in FP32 precision.

The scaling factor $1/r_i$ can be applied in two ways:

1. **Post-multiplication scaling**: Compute $\mathbf{T} = (\mathbf{P}_i)_{\text{BF16}}(\mathbf{V}_i)_{\text{BF16}}$ first, then scale by $1/r_i$. This requires scalar-matrix multiplication in FP32, which has to be executed on the Vector core and incurs significant memory access overhead.
2. **Pre-multiplication scaling**: Scale $\mathbf{P}_i$ by $1/r_i$ first, then perform the matrix multiplication $\left(\mathbf{P}_i/r_i\right)_{\text{BF16}} \left(\mathbf{V}_i\right)_{\text{BF16}}$. This approach, while more straightforward to implement, amplifies rounding error: since $1/r_i$ is cast to BF16, the term $\left(1/r_i\right)_{\text{BF16}}$ introduces quantization error even when $\mathbf{P}_i$ contains values as benign as 1 (which is exactly representable in BF16). This error propagates multiplicatively and significantly degrades output accuracy.

To mitigate this, we reformulate the computation by defining $1/r_i' = \left(1/r_i\right)_{\text{BF16}}$, leading to:

$$\tilde{\mathbf{O}}_i = \tilde{\mathbf{O}}_{i-1} \times 2^{n_i - n_{i-1}} + \frac{r_i'}{r_i} \left(\frac{\mathbf{P}_i}{r_i'}\right)_{\text{BF16}} \left(\mathbf{V}_i\right)_{\text{BF16}}. \tag{12}$$

Let $c_i = r_i/r_i'$ and $\bar{\mathbf{O}}_i = \tilde{\mathbf{O}}_i \cdot c_i$. Substituting yields the compensated recurrence:

$$\bar{\mathbf{O}}_i = \bar{\mathbf{O}}_{i-1} \times 2^{n_i - n_{i-1}} \times \frac{c_i}{c_{i-1}} + \left(\frac{\mathbf{P}_i}{r_i'}\right)_{\text{BF16}} \left(\mathbf{V}_i\right)_{\text{BF16}}. \tag{13}$$

This formulation requires scaling $\bar{\mathbf{O}}_{i-1}$ by both $2^{n_i - n_{i-1}}$ and c_i/c_{i-1}. The former is efficiently handled via **AtomicAdd** (cf. Lemma 3.1); the latter is non-trivial but tractable due to the proximity of c_i/c_{i-1} to 1. Since BF16 has a relative precision of approximately $1/256$, we have $c_i/c_{i-1} = 1 + \epsilon$ with $|\epsilon| < 1/256$.

Considering a floating-point number F,

$$F = (-1)^S \left(1 + \frac{M}{2^{23}}\right) 2^{E-127} \tag{14}$$

Multiplying F by $1 + \epsilon$ yields:

$$F \cdot (1+\epsilon) = (-1)^S \cdot 2^{E-127} \cdot \left(1 + \frac{M + 2^{23}\epsilon + \epsilon M}{2^{23}}\right). \tag{15}$$

To approximate the integer representation of the result, we estimate $M \approx 2^{22}$ (the midpoint of the mantissa range), leading to:

$$\begin{aligned} \text{AS\_INT32}(F \cdot (1+\epsilon)) &\approx \text{AS\_INT32}(F) + \text{round}\left(2^{23}\epsilon + 2^{22}\epsilon\right) \\ &= \text{AS\_INT32}(F) + \text{round}\left(1.5 \cdot 2^{23}\epsilon\right). \end{aligned}$$

This additive integer adjustment enables efficient implementation via atomic operations without invoking costly FP32 multiplies. Combined with Eq. (9), this yields the final Algorithm 2.

Critically, this compensation introduces only minimal overhead confined to the $[V_1]$ stage and has negligible impact on overall kernel performance while substantially improving numerical fidelity.

B Proofs for the Preload Pipeline

B.1 Optimization Analysis of the Preload Pipeline

As defined in Section 4.1.2, each internal dependency chain established within a Cycle reduces the required Preload count by one. For instance, in Fig. 6a, if $[V_1]$ can directly consume the result of $[C_1]$ computed within the same Cycle (i.e., $[C_1] \to [V_1]$ is internally resolved), then $[C_1]$ need not be precomputed in the Preload phase. This internal dependency decouples $[V_1]$ from its external dependency on $[C_1]$, thereby reducing the Preload burden.

Importantly, later stages in the dependency chain (e.g., $[V_2]$, $[V_3]$) typically require longer prerequisite sequences to be internally satisfied. Hence, resolving dependencies for later blocks, such as $[C_2] \to [V_2]$ in Fig. 6b, yields greater Preload reduction (e.g., eliminating 3 blocks instead of 1). This observation suggests that **maximizing the number of internal dependency chains** is key to minimizing the Preload overhead.

Lemma B.1. *For a stage consisting of n CV pairs, the total length of the sequential dependency chain is $2n-1$. Let s denote the number of internal dependency chains constructed within a Cycle. Then the Preload count is given by:*

$$\textit{Preload count} = (2n-1) - s.$$

Each internal chain reduces the need for external precomputation, so a larger s leads to a smaller Preload count.

This leads to the central question: Under the constraint of preserving pairwise CV dependencies, does there exist a universal upper bound $\bar{s}$ on the number of internal dependency chains that can be achieved via rearrangement, regardless of individual stage durations? If so, what is $\bar{s}$?

B.2 Problem Construction and Upper Bound Derivation

To address this, we consider two cases based on the relative total durations of Cube and Vector stages: $\sum[V_i] \le \sum[C_i]$ (Cube-dominated) and $\sum[V_i] \ge \sum[C_i]$ (Vector-dominated). We first analyze the Cube-dominated case.

Consider n CV pairs forming the chain $[C_1] \to [V_1] \to [C_2] \to [V_2] \to \cdots \to [C_n] \to [V_n]$. We aim to prove that no pipeline can guarantee more than $\bar{s} = n-1$ internal dependency chains for arbitrary stage durations.

To establish this upper bound, we construct an adversarial scenario: suppose there exists a Vector stage $[V_k]$ such that

$$[V_k] + [C_j] > \sum_{i=1}^{n} [C_i], \quad \forall j \in \{1, 2, \ldots, n\}. \tag{16}$$

This implies that $[V_k]$ cannot overlap with any Cube stage without exceeding the total cube duration. Given that all Vector stages must be overlapped by the cumulative Cube execution (to avoid extending the Cycle), we deduce:

- No Cube stage can complete before $[V_k]$ starts — otherwise, for any such completed Cube stage $[C_{k'}]$, we have $[V_k] + [C_{k'}] > \sum[C_i]$, violating the overlap constraint. Thus, the $k-1$ Vector stages preceding $[V_k]$ cannot depend on their corresponding Cube stage.
- No new Cube stage can start after $[V_k]$ ends, so the $n-k$ Vector stages following $[V_k]$ cannot be consumed by their Cube stages.
- $[V_k]$ itself can neither depend on any Cube stage nor be consumed by any Cube stage.

Consequently, at most $(k-1) + (n-k) = n-1$ internal dependency chains can be formed.

By Lemma B.1, the minimal Preload count in this case is $(2n-1) - (n-1) = n$. The symmetric case ($\sum[V_i] \ge \sum[C_i]$) yields the same bound via analogous reasoning. Thus, we conclude:

Lemma B.2. *For any n CV pairs satisfying the sequential dependency chain, no pipeline can universally guarantee more than $n-1$ internal dependency chains, regardless of stage durations or scheduling permutations.*

B.3 Existence of Optimal Pipeline: Formalization and Key Theorem

We now prove that a pipeline achieving exactly $n-1$ internal dependency chains always exists under $\sum[V_i] \leq \sum[C_i]$. Without loss of generality, consider $n = 3$. The six permutations of $\{[C_1], [C_2], [C_3]\}$ fall into two cyclic groups:

- Group 1 (clockwise): $[C_1][C_2][C_3]$, $[C_2][C_3][C_1]$, $[C_3][C_1][C_2]$
- Group 2 (counter-clockwise): $[C_1][C_3][C_2]$, $[C_3][C_2][C_1]$, $[C_2][C_1][C_3]$

Focusing on Group 1 (see Fig. 11), each permutation admits two internal $[V] \to [C]$ dependency chains. If at least one permutation satisfies its temporal constraints, then a valid pipeline with $n-1=2$ chains exists. The same holds for Group 2, implying existence for $n=3$.

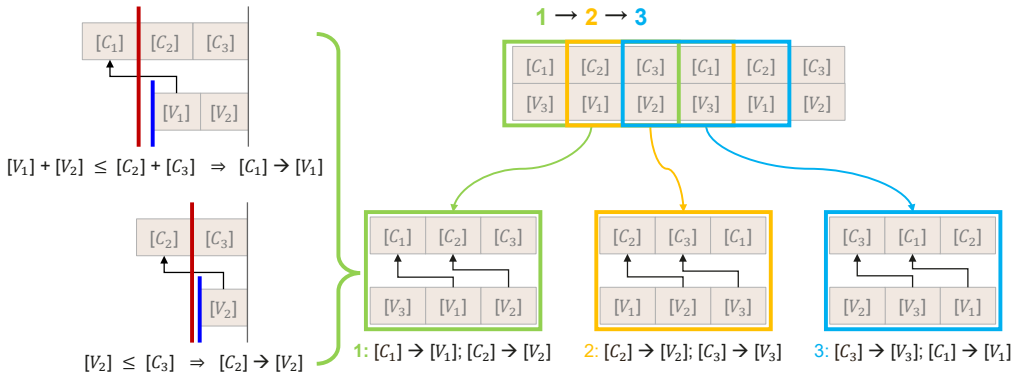

Figure 11: Cyclic permutations of $[C_1][C_2][C_3]$ with two internal $[V] \to [C]$ dependency chains and their temporal constraints.

For permutation $[C_1][C_2][C_3]$, enforcing $[C_1] \to [V_1]$ and $[C_2] \to [V_2]$ requires:

$$[V_1] + [V_2] \leq [C_2] + [C_3] \quad \text{and} \quad [V_2] \leq [C_3].$$

Similarly, the other two permutations in Group 1 yield:

1. $[V_1] + [V_2] \leq [C_2] + [C_3]$ and $[V_2] \leq [C_3]$: enables $[C_1] \to [V_1]$, $[C_2] \to [V_2]$;
2. $[V_2] + [V_3] \leq [C_3] + [C_1]$ and $[V_3] \leq [C_1]$: enables $[C_2] \to [V_2]$, $[C_3] \to [V_3]$;
3. $[V_3] + [V_1] \leq [C_1] + [C_2]$ and $[V_1] \leq [C_2]$: enables $[C_3] \to [V_3]$, $[C_1] \to [V_1]$.

We must show that at least one of these three compound conditions holds.

Generalizing to n CV pairs, we formalize the existence guarantee as:

Theorem B.1. *Given n CV pairs with dependency chain $[C_1] \to [V_1] \to \cdots \to [C_n] \to [V_n]$ and $\sum_{i=1}^{n}[V_i] \leq \sum_{i=1}^{n}[C_i]$, there exists an index $k \in \{1, \ldots, n\}$ such that:*

$$\bigwedge_{j=1}^{n-1} \left(\sum_{i=0}^{j-1} [V_{n-k-i}] \leq \sum_{i=0}^{j-1} [C_{n+1-k-i}] \right), \tag{17}$$

where indices are interpreted cyclically modulo n (i.e., $[V_{n+i}] \equiv [V_i]$, $[C_{-i}] \equiv [C_{n-i}]$). This guarantees the existence of a pipeline with exactly $n-1$ internal dependency chains.

B.4 Proof of Theorem B.1

To simplify the analysis, define an auxiliary sequence $\{a_i\}_{i=1}^{n}$:

$$a_i = [V_i] - [C_{i+1}], \quad \text{with} \quad [C_{n+1}] \equiv [C_1]. \tag{18}$$

The global constraint becomes:

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} [V_i] - \sum_{i=1}^{n} [C_i] \leq 0. \tag{19}$$

B.4.1 Reformulation via Partial Sums

Let $m = n - k$. Then Eq. (17) is equivalent to:

$$\sum_{i=0}^{j-1} a_{m-i} \leq 0, \quad \forall j \in \{1, \ldots, n-1\}, \tag{20}$$

with cyclic indexing. Thus, proving the above Theorem B.1 reduces to showing that some $m \in \{1, \ldots, n\}$ satisfies Eq. (20) for all j.

B.4.2 Key Insight: Minimum Partial Sum

Define the partial sum $F(l) = \sum_{i=1}^{l} a_i$ for $l = 0, 1, \ldots, n$, with $F(0) = 0$. By Eq. (19), $F(n) \leq 0$. Let $k \in \{1, \ldots, n\}$ be such that $F(k)$ is minimal.

We claim $m = k$ satisfies Eq. (20). Consider two cases:

Case 1: $1 \leq j \leq k$

$$\sum_{i=0}^{j-1} a_{k-i} = F(k) - F(k-j) \leq 0, \tag{21}$$

since $F(k) \leq F(k-j)$ by minimality.

Case 2: $k < j \leq n-1$ Using cyclicity $a_t = a_{t+n}$:

$$\begin{aligned} \sum_{i=0}^{j-1} a_{k-i} &= \sum_{i=0}^{j-1} a_{k-i+n} \\ &= F(k+n) - F(k+n-j) \\ &= [F(n) + F(k)] - F(k+n-j) \qquad (22) \\ &\leq F(n) \leq 0, \qquad (23) \end{aligned}$$

since $F(k+n-j) \geq F(k)$ and $F(n) \leq 0$.

B.4.3 Conclusion

In both cases, the inequality holds. Therefore, setting $m = k$ (i.e., choosing the permutation aligned with the minimum partial sum) guarantees the existence of $n-1$ internal dependency chains. For the symmetric case $\sum[V_i] \geq \sum[C_i]$, the same logic applies by reversing roles and inequalities. This completes the proof of Theorem B.1.