Tackling the Dynamicity in a Production LLM Serving System with SOTA Optimizations via Hybrid Prefill/Decode/Verify Scheduling on Efficient Meta-kernels

Mingcong Song* *Huawei*
Xinru Tang* *Tsinghua University*
Fengfan Hou *Huawei*
Jing Li *Huawei*
Wei Wei *Huawei*
Yipeng Ma *Huawei*
Runqiu Xiao *Huawei*
Hongjie Si *Huawei*
Dingcheng Jiang *Tsinghua University*
Shouyi Yin *Tsinghua University/Shanghai AI Lab*
Yang Hu *Tsinghua University*
Guoping Long *Huawei*

Abstract

Meeting growing demands for low latency and cost efficiency in production-grade large language model (LLM) serving systems requires integrating advanced optimization techniques. However, dynamic and unpredictable input-output lengths of LLM, compounded by these optimizations, exacerbate the issues of workload variability, making it difficult to maintain high efficiency on AI accelerators, especially DSAs with tile-based programming models. To address this challenge, we introduce XY-Serve, a versatile, Ascend native, end-to-end production LLM-serving system. The core idea is an abstraction mechanism that smooths out the workload variability by decomposing computations into unified, hardware-friendly, fine-grained meta primitives. For attention, we propose a meta-kernel that computes the basic pattern of matmul-softmax-matmul with architectural-aware tile sizes. For GEMM, we introduce a virtual padding scheme that adapts to dynamic shape changes while using highly efficient GEMM primitives with assorted fixed tile sizes. XY-Serve sits harmoniously with vLLM. Experimental results show up to 89% end-to-end throughput improvement compared with current publicly available baselines on Ascend NPUs. Additionally, our approach outperforms existing GEMM (average 14.6% faster) and attention (average 21.5% faster) kernels relative to existing libraries. While the work is Ascend native, we believe the approach can be readily applicable to SIMT architectures as well.

1 Introduction

Large language models (LLMs) [41, 42] have achieved impressive accuracy and are widely applied in fields like natural language processing [19] and computer vision [21, 25]. As shown in Fig. 1(a), LLM inference typically consists of two stages: prefill and decode. During the prefill stage, LLMs process the user's input to generate an initial token, while concurrently caching the key/value (K/V) data for future use.

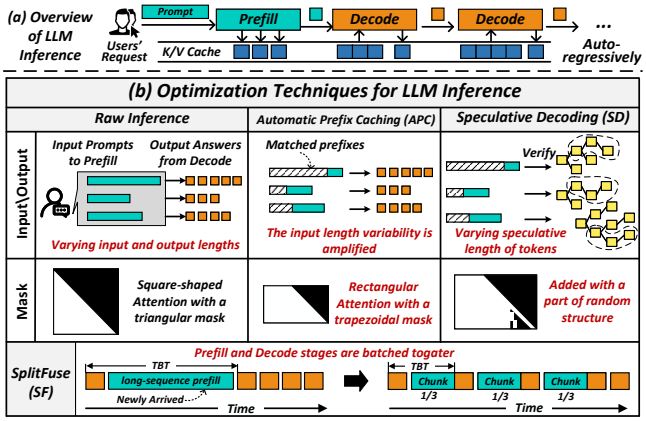

Figure 1: Dynamics of LLM Inference.

In the decode stage, tokens are generated sequentially in an auto-regressive manner. Despite their impressive performance, LLMs come with significant computational costs and latency. As the model size increases and input sequences become longer, the computational demands grow substantially, making online inference increasingly challenging [30].

To address these challenges, a number of optimization techniques have emerged to reduce inference costs and latency, such as Automatic Prefix Caching (APC) [8, 47], Speculative Decoding (SD) [20, 22, 32, 33, 38, 46], and SplitFuse [17, 18, 26]. APC enables new queries with matching prefixes to reuse cached K/V data and skip computations for shared segments, thereby improving prefill performance. The adoption of draft model in SD provides a chance for target model to generate multiple tokens per step, enhancing key/value cache and model weight reuse, which helps mitigate the memory-bound bottleneck of decode in draft model. SplitFuse splits long-sequence prefill tokens into smaller chunks and schedules them alongside decode tokens, reducing interruptions to the decode stage.

While these optimizations promise to improve inference efficiency, they also introduce new complexities. For LLM

*These authors contributed equally to this work.

serving of a production system, the key challenge is how to integrate all these optimizations efficiently on AI accelerators given that a friendly SIMT programming model is lacked. We illustrate this issue in Fig. 1(b). Performance already struggles with unpredictable and varying input/output token lengths, these optimizations can further exacerbate the issue.

For example, APC increases the variability in input prompt lengths, as the number of cached prefixes depends on both query history and real-time memory availability. With SD, the decode stage no longer processes one token at a time. Instead, it handles a dynamically varying speculative length of tokens [37]. This new stage is referred to as the Verify stage [4, 45]. SD also transforms the attention mask from a standard causal mask into a more complex, dynamically generated version. Additionally, SplitFuse combines the prefill and decode stages into a single batch, further complicating the management of multiple stages within one batch.

These dynamicities pose significant challenges for LLM computations, particularly in Linear and Attention modules. First, the uncertainty of input lengths leads to arbitrary matrix shapes in Linear ops, complicating optimization efforts aimed at achieving peak computational efficiency. Second, the adoption of technologies like APC, SD, and SplitFuse introduces greater diversity in attention shapes and mask structures, weakening the effectiveness of existing attention kernels.

Furthermore, in practical systems, the Prefill (P), Decode (D), and Verify (V) stages may operate independently or in combination. Even in disaggregated deployments, the D node may run both D and V stages simultaneously. If disaggregated deployment [27, 28, 39, 48] nodes support dynamic role switching, the hybrid P/D/V combinations issue may also arise during role transitions. Since these stages present varying computational loads during the attention phase, attempting to enumerate and optimize for every possible stage combination becomes a labor-intensive and impractical task.

To tackle the challenges posed by dynamicities, we present XY-Serve, a versatile, Ascend native, and end-to-end production LLM-serving system. The main idea is to introduce an abstraction to bridge the gap between varying high level workloads and fixed hardware-friendly low level meta-primitives. Specifically, XY-Serve features a token-wise scheduling mechanism that batches tokens in chunks. Tokens could be from either prefill, verify, and decode stages. Then the token chunks will be processed by three core components: workload decomposition, computation task reordering, and meta kernels.

Workload decomposition is a mechanism to decompose and map dynamic workloads onto hardware-friendly meta-primitives. For attention computations, it unifies the P/D/V stages through dynamic tiling, generating hardware-friendly tile-based computational tasks. Each task computes a basic matmul-softmax-matmul pattern. For GEMM, it decomposes dynamic-shaped matmul ops into a small set of basic matmul primitives with fixed tile sizes. This transformation seems deceptively straightforward but is surprisingly hard to implement without any actual padding overhead. We introduce a novel virtual padding mechanism to address this issue without introducing any extra overhead on AI accelerators with tile-based programming models.

After decomposition, computation task reordering reorganizes decomposed tasks and schedules them for hardware processing cores. The goal of reordering is to smooth out varying task sizes at fine-grained level, balancing workload on different cores for high execution efficiency. For attention, P/D/V tasks have drastically different granularity; thus, reordering of decomposed tasks is essential to achieve load balance. The case for GEMM is different; task reordering is an effective approach to improve L2 cache locality and mitigate bank conflicts.

With workload decomposition and task reordering, it is possible to construct efficient kernels for attention and GEMM. For attention, we propose meta kernels to efficiently parallelize computations of matmul-softmax-matmul patterns with different tile sizes, without needing to differentiate which P/D/V stage they originate from. With this decoupling approach, it is feasible to integrate a range of optimizations seamlessly and efficiently, including APC, PageAttention [31], SplitFuse, SD, and FlashAttention [24]. These optimizations work synergistically within our meta kernels, achieving superior performance.

Moreover, our attention meta kernels come with aggressive on-chip Cube Core-Vector Core orchestration pipelines, together with novel schemes to handle various kinds of attention masks dynamically. These low level techniques minimize off-chip memory accesses and maximize on-chip workload balance and execution efficiency. For GEMM ops, we first design and implement a set of highly efficient meta kernels with fixed tile sizes. To handle arbitrary input shapes dynamically, we employ virtual padding at the on-chip memory level, coupled with selective HBM reads and writes, and do not introduce any actual padding overhead. In other words, our approach allows matrix computations to seamlessly handle dynamic shapes while preserving the performance benefits of fixed-shape optimizations.

Experimental evaluation shows that our attention kernels achieve higher efficiency in production workloads, outperforming existing implementations(torch-npu PFA [12] and IFA [11]) by on average of 21.5%. Our GEMM kernels not only support arbitrary matrix shapes but also improve performance by on average of 14.6% over existing baselines. In end-to-end evaluation, XY-Serve achieves improvement up to 89% over Ascend-vLLM [14] on publicly available datasets [13]. While currently implemented on Ascend NPUs [34, 35], these techniques can be applied to other AI platforms as well. Finally, we conduct a comparative evaluation against GPU-based inference systems. In terms of end-to-end MFU and MBU, XY-Serve performs on par with the best GPU implementations.

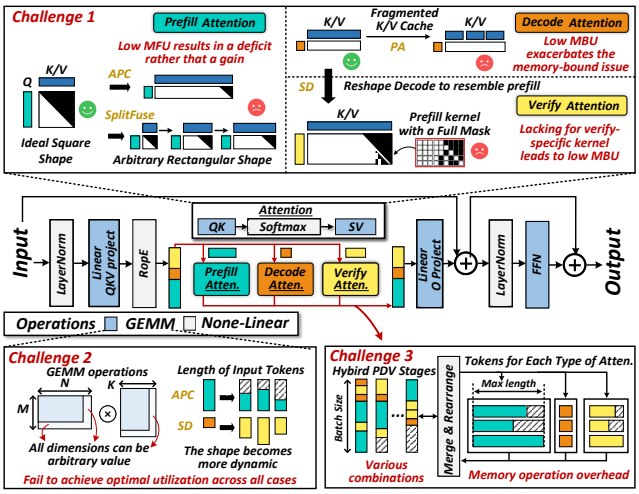

Figure 2: Challenges Posed by Dynamic Workloads.

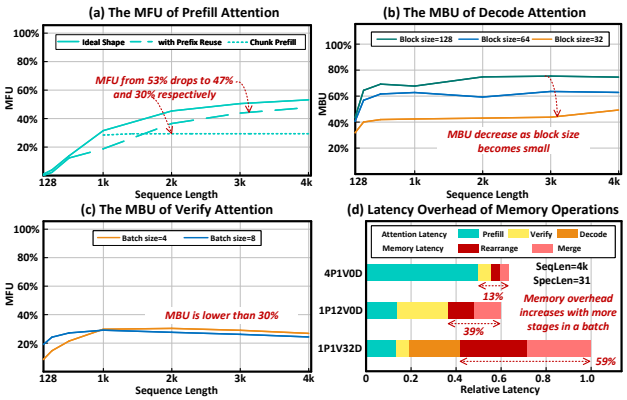

Figure 3: The MFU and MBU of Attention Kernel.

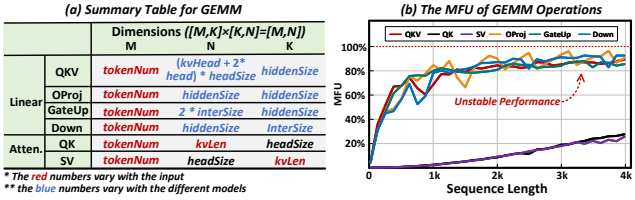

Figure 4: GEMM Operations in LLMs and their MFU.

2 Motivation

In this section, we explore the challenges posed by dynamic workloads in Linear and Attention modules and analyze the complexities of handling P/D/V hybrid stages, shown in Fig. 2. Finally, we discuss the additional challenges faced by AI accelerators with tile-based programming models, such as the Huawei Ascend NPU [34, 35], in supporting dynamic workloads.

2.1 Diverse Attention

The introduction of new technologies, such as APC, SD, and SplitFuse, increases the diversity of Attention shapes and mask structures, leading to MFU and MBU issues of the current NPU attention kernel (torch-npu 2.1 FusedInferAttentionScore [10]), as illustrated in Fig 3.

Firstly, for prefill Attention, without any optimizations, the query length equals the key-value length, resulting in a square-shaped Attention score matrix and a lower triangular mask. This shape is ideal for optimization [36], as the sparsity in the mask can be exploited to achieve high performance. The current state-of-the-art (SOTA) torch-npu kernel efficiently handles such square-shaped inputs with triangular masks, achieving an MFU of 53%.

However, in real-world scenarios involving prefixes, parts of the key and value are reused from the K/V cache [8, 47], transforming the Attention score shape from a square to a rectangle with arbitrary dimensions. For these rectangular shapes, the performance of the torch-npu kernel degrades significantly, with MFU dropping to 47% and 30%, respectively. While the reuse of the K/V cache theoretically reduces computation, the decline in kernel efficiency offsets this benefit, resulting in no meaningful end-to-end performance improvement.

Secondly, unlike Linear, where tokens from different batches can share weights, decode Attention requires each token to have its own independent K/V cache, making reuse impossible. This leads to inherently memory-bounded computations. PagedAttention (PA) [31] further fragments the K/V cache access patterns, limiting access to small block-sized portions instead of large contiguous blocks. As illustrated in Fig. 3(b), the MBU decreases as block size becomes small, exacerbating the memory bottleneck of the decode stage.

Lastly, Speculative Decoding (SD) [20, 33, 38] reshapes decode Attention to resemble prefill Attention but with a shorter query length. Different SD algorithms generate tokens with varying causal structures, resulting in diverse masks during the verify stage. However, due to the absence of a verify-specific kernel for Ascend, the prefill Attention kernel—equipped with full masks—is used as a fallback. This fallback prevents verify-specific optimizations, leading to MBU lower than 30%.

2.2 Dynamic GEMM

GEMM is a core operation in LLMs. Beyond the four GEMM operations in the Linear section, the query-key (QK) and score-value (SV) computations in Attention also rely on GEMM. We summarize all of GEMM operations in Fig. 4(a). For Linear operations, the M dimension is tied to the number of tokens. For Attention QK and SV operations, the N and K dimensions are dependent on the token K/V length. As previously discussed, the number of tokens in the prefill and verify stages is highly dynamic, and this variability is further

amplified by the introduction of Automatic Prefix Caching (APC) [8, 47], making token lengths even more unpredictable.

In addition, for Linear GEMM, the dimensions N and K are influenced by *hiddenSize*, which varies with the architecture of the LLM. Consequently, the M, N, and K dimensions in GEMM operations for LLM are all dynamically changing.

Designing a matrix multiplication on AI accelerators to support arbitrary shapes while ensuring high performance is inherently challenging. Most accelerators are typically optimized for specific tile sizes, such as 16×16, making it difficult to fully utilize their capabilities with arbitrary shapes. Irregular shapes complicate parallelism and load balancing, causing inefficiencies in workload distribution across processing units. The requirement for flexible algorithms to adapt to diverse shapes increases algorithmic complexity. Additionally, handling boundary conditions for non-uniform matrix sizes introduces overhead, further affecting performance. As shown in Fig. 4(b), employing a general-purpose GEMM kernel (torch-npu 2.1 linear operator [9]) to accommodate all possible results in unstable performance and fails to achieve optimal utilization across all GEMM operations.

2.3 Hybird P/D/V Stages

In practical systems, P/D/V stages may exist independently or coexist simultaneously, leading to arbitrary combinations of interleaved P/D/V stages within a given scheduling budget. For Linear operations, tokens from different stages can be grouped together and treated as the left matrix in a GEMM operation, sharing the same weight matrix on the right. This approach is straightforward, as tokens from different stages can reuse the same large model weights.

In contrast, handling Attention operations is significantly more complex. Stages are independent, and even within the same stage, batches are also independent. Enumerating and tailoring optimizations for each possible combination of stages and batches is highly labor-intensive and impractical.

A common alternative is a batch-by-batch execution, such as selective batching [44], which processes each batch independently by invoking the corresponding Attention kernel. However, this method introduces additional memory overhead from splitting, rearranging, and merging data, which degrades overall system performance. As shown in Fig. 3(d), the memory overhead may account for more than 50%. Furthermore, batch-by-batch processing leads to inefficient utilization of computational resources, further limiting system efficiency.

2.4 Ascend NPU Micro-architecture

Built on Huawei's DaVinci architecture [34, 35], the Huawei Ascend NPU is a high-performance AI processor. Fig. 5 illustrates the micro-architecture of the Ascend, which consists primarily of AIC and AIV components. The AIC, similar to Nvidia's Tensor Core, handles matrix computations, while

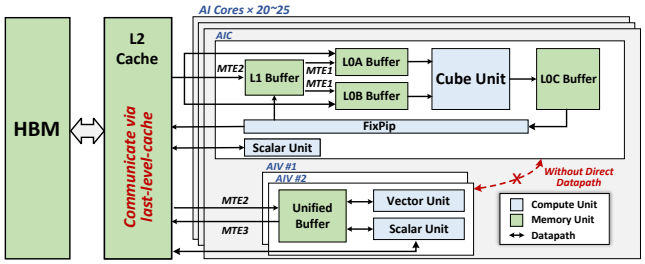

Figure 5: The Micro-architecture of Ascend 910B.

the AIV is responsible for vector operations. AIC and AIV are separated without a direct datapath, so ensuring their data interactions occur via the L2 cache is crucial when designing mixed kernels. Compared to GPUs, NPUs have larger core granularity, making load balancing between cores even more critical. The Memory Transfer Engine (MTE) handles data movement. Whether AIC, AIV, or MTE, all data processing and transferring occur at the tile level. AI accelerators with tile-based programming models are gaining increasing attention in the community. However, this tile-based processing model encounters significant challenges when managing dynamic workloads that vary at the token granularity. Typical solutions involve complex padding/unpadding operations, which result in wasted computation and memory, leading to substantial overhead.

3 Overview of XY-Serve

To address the aforementioned challenges, we developed XY-Serve, a versatile end-to-end production LLM-serving system, which is built on four key components: Token-wise Scheduling, Dynamic Task Decomposition and Reordering, Meta-Attention, and SmoothGEMM.

3.1 Token-wise Scheduling

When a user's request enters the system, it first passes through the APC module, which matches the incoming prompt against existing prompts in the K/V cache, enabling token-wise reuse. Any unmatched tokens are added to the scheduling queues. Consequently, the prompt length in the scheduling queues is the user's input length minus the length of tokens already cached in the K/V cache. Since both the user's input and the cached token lengths are dynamically variable, the prompt lengths in the scheduling queues become even more dynamic. The scheduling queues also include decode and speculative tokens from previous requests awaiting processing.

Our scheduling system selects a fixed-budget length of tokens from the scheduling queues to form chunks, which may include tokens from prefill, verify, or decode stages. To improve first-token latency, prefill requests are prioritized. If a user's prefill prompt exceeds the budget length, it is split into

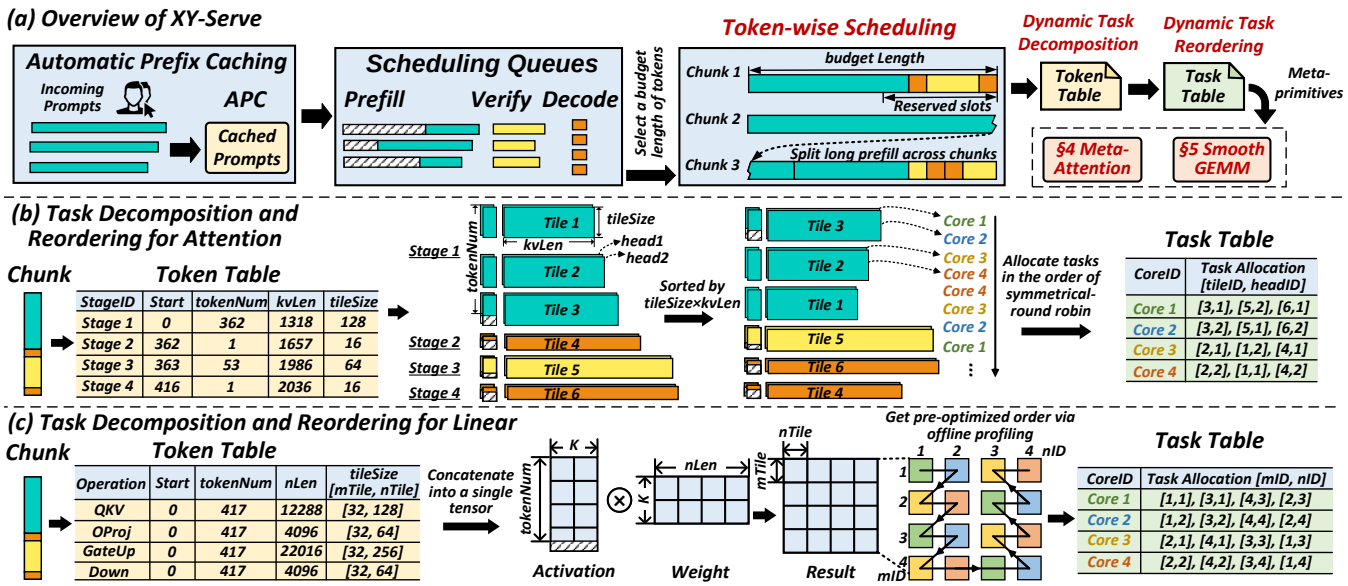

Figure 6: Overview of XY-Serve.

smaller parts to ensure each scheduled chunk remains within the budget. To minimize interruptions caused by prefill on decode and maintain a stable Time Between Tokens (TBT), certain slots are reserved for decode and speculative tokens. Prefill-only chunks are scheduled only when the queue has no decode or speculative tokens.

As shown in Fig. 6(a), the composition of P/D/V stages within a single chunk is inherently unpredictable, with token counts for each stage varying arbitrarily and each stage having a distinct historical K/V length. Additionally, the total token count in a chunk may not always match the budgeted length and can fall below the budget under a low system load. To address the four levels of dynamism, our scheduling operates entirely at the granularity of individual tokens without concern for their origin.

3.2 Task Decomposition

While token-wise scheduling improves efficiency by reducing bubbles and optimizing resource utilization, the four levels of dynamism it introduces pose significant challenges for execution, especially on AI accelerators with tile-based programming models.

To address these challenges, we propose a dynamic decomposition mechanism that converts dynamic workloads into hardware-friendly, tile-based computational units. Using the Token-Table, each stage is logically decomposed into tile blocks. At the tile level, computation modules can process these blocks in parallel without distinguishing their P/D/V stage origin. Importantly, this tiling decomposition is purely logical, requiring no changes to the physical data layout.

In the following sections, we present a detailed analysis of how tiling decomposition is applied to Attention and Linear layers.

3.2.1 Attention Decomposition

For attention, the Token-Table contains entries for each P/D/V stage, with each entry specifying key attributes, including the *stageID*, the *start position*, the number of *tokenNum*, the historical *kvLen*, and the *tileSize*. As shown in Fig. 6(b), stage-1 (P) is decomposed into three tiling blocks, while stage-2 and stage-4 (D) are each divided into one tiling block, and stage-3 (V) is also decomposed into one tiling block. Each tiling block consists of *headNum* tiling units, resulting in a total of $6 \times headNum$ tiling units at the tiling level.

3.2.2 Linear Decomposition

For Linear operations, since tokens from different stages can share the same weights, they are concatenated into a single, large tensor and multiplied by the shared weights. This approach avoids multiple GEMM invocations, enhances weight reuse, and streamlines computation. Consequently, Linear operations do not need to be differentiated between stages, and can be processed uniformly.

In the Token-Table, each Linear operator corresponds to a single entry shown in 6(c). The four primary Linear operations are *QKV*, *OProj*, *GateUp*, and *Down*. For these operations, the *start position* is set to 0, indicating that all tokens are processed from the beginning of the concatenated tensor. The *tokenNum* equals the total number of tokens in the currently scheduled chunk. Tiling is performed on the result matrix of dimensions $tokenNum \times nLen$, where each tiling block corresponds to a submatrix of the result. Each tiling block

is assigned to a single AI core, allowing different cores to process distinct blocks in parallel. Each Linear operator has its own specific *tileSize*, determined by the *tokenNum* and *nLen* shapes of the operation.

3.3 Task Reordering

After decomposing the dynamic workloads from P/D/V mixed stages into fundamental tile units, it is necessary to reorder these tile units and generate a Task-Table to enhance performance. The Task-Table is responsible for scheduling these tile units onto the hardware, with each entry specifying a *coreID* and the list of tiles assigned to that core. Based on this Task-Table, Attention, and Linear can simply retrieve the corresponding tile units according to their *coreID*. This approach not only maximizes hardware efficiency but also simplifies the design of Attention and Linear kernels.

3.3.1 Attention Reordering

After performing dynamic tiling on the various stages, the resulting tiles exhibit varying values for *tileSize* and *kvLen*. This variation can lead to load imbalances during parallel processing. To address this, we calculate the computational load of each tile as its area, defined as $tileSize \times kvLen$.

For efficient scheduling, the tiles are initially sorted based on their computational load, from largest to smallest. Subsequently, the tiles are allocated to the AI cores in a symmetrical round-robin fashion. As depicted in Fig. 6(b), assuming there are four AI cores, the tile units are assigned in the sequence core-1, core-2, core-3, core-4, core-4, core-3, core-2, core-1, and so forth, repeating this pattern to ensure a balanced and efficient allocation of computational tasks.

This task scheduling information is stored in the Task-Table and passed to the Attention module. The Task-Table guides the Attention module to perform parallel processing efficiently, leveraging both the head and tile dimensions. This mechanism ensures balanced computation across AI cores, maximizing hardware utilization.

3.3.2 Linear Reordering

Given that only a limited set of fixed shapes is supported, we perform offline optimization to determine the most efficient task allocation strategies for linear ops. This involves profiling and customizing task allocation for each shape to maximize performance. The optimized strategies are stored for use during execution. During runtime, XY-Serve leverages the Token-Table to identify the current shape and uses this information to retrieve the corresponding pre-optimized Task-Table. The Task-Table is then passed to the Linear module, guiding it to execute tasks in an optimized manner.

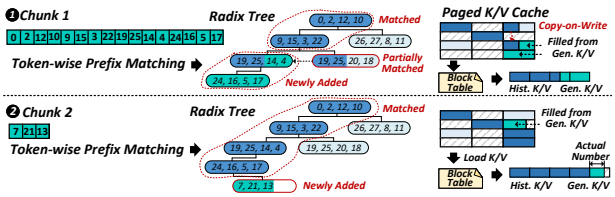

Figure 7: Token-wise K/V Cache Reuse.

4 Meta-Attention

In this section, we first explain how our attention module supports advanced features such as APC, Chunked Prefill, and SD. Then, we describe how we optimize attention performance to push it to the hardware limits.

4.1 Meta-Attention Design

4.1.1 Handling Token-wise Processing

The core requirement for supporting both prefix reuse and Chunked Prefill is that the attention module must be capable of handling arbitrary K/V cache lengths and performing token-wise K/V cache reuse. To achieve this, we use a radix tree to efficiently manage the K/V cache. As shown in Fig. 7, each node in the radix tree represents a K/V cache block. The radix tree allows for quick matching of historical K/V cache blocks. When a mismatch occurs at a particular node (i.e., its corresponding K/V cache block contains only a partial match), we use a copy-on-write mechanism to create a new block, refresh the new data into this block, and then add the block back into the radix tree. If additional new blocks are generated after this mismatch, these blocks are directly inserted as child nodes of the mismatched block in the radix tree.

This mechanism effectively manages both historical and newly generated K/V data, seamlessly merging them using copy-on-write to ensure the continuity of the K/V cache. During the prefill attention process, the corresponding K/V blocks—both historical and newly generated—are read based on the block table. We also track the actual number of tokens in the last block, ensuring accurate token-wise processing.

Additionally, our system can automatically cache historical K/V data as prefixes, relieving the user from manually specifying them. The system allows users to set an upper limit on the amount of historical K/V data to be cached. Once this limit is reached, or when space is needed for new K/V data, the system will automatically evict the least recently used K/V blocks from the leaf nodes of the radix tree.

4.1.2 Minimizing Mask for Speculative Decoding

Speculative execution can be classified into two types: sequence-based speculation [20, 33, 38, 46] and tree-based speculation [22, 32]. Sequence-based speculation generates

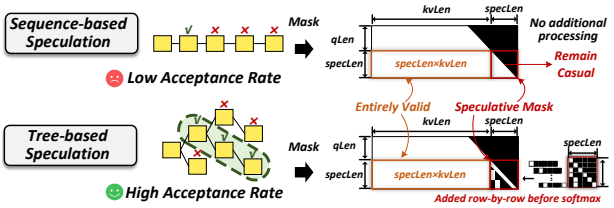

Figure 8: Speculative Decoding Algorithms.

multiple tokens within a single sequence; however, its acceptance rate is generally low. In contrast, tree-based speculative algorithms generate predictions for multiple sequences simultaneously, organizing them in a tree structure. This approach can further improve the acceptance rate of speculation.

Both types need support for arbitrary speculation lengths. Furthermore, tree-based speculation utilizes a more complex and dynamic mask, which is not well-suited for vector-based hardware. By analyzing the structure of this mask, we can identify regularities that enable efficient processing.

As illustrated in Fig 8, the speculative decoding extends the causal mask of standard prefill ($qLen \times kvLen$) by introducing a $specLen \times (kvLen + specLen)$ region. Within this region, the $specLen \times kvLen$ is entirely valid, while only the $specLen \times specLen$ section requires special handling, referred to as the 'Speculative Mask'.

For sequence-based speculation, the speculative mask remains causal, and no additional processing is required, allowing the direct application of our mask-free approach. In tree-based speculation, we generate only the $specLen \times specLen$ part of the mask externally, which is then passed to the kernel and applied to the corresponding attention score matrix.

Our design processes the speculative mask row-by-row, enabling precise control over the start position and length of the mask for each row. This method efficiently supports arbitrary speculation lengths. Once the mask is adjusted, the subsequent computation follows the standard prefill process. Using the speculative mask as a mediator, we can efficiently and seamlessly support a wide range of speculative algorithms.

4.2 Meta-Attention Optimizations

4.2.1 Tile-Based Cube-Vector Orchestration

To achieve parallel execution of cube and vector units, we propose a pipeline, shown in Fig. 9. It ensures intermediate data transfers occur exclusively via the L2 cache, avoiding costly HBM accesses. The cube unit is responsible for the QK and SV computations, while the vector unit performs the $Softmax$. If processed sequentially ($QK \rightarrow Softmax \rightarrow SV$), only one unit would be active at a time, leading to inefficiencies. To address this, we adopted a pipelined approach as displayed in Fig. 10(a): after completing the QK computation for the first tiling data, its $Softmax$ computation is initiated while

Optimal Pipeline Strategies of Meta-Attention

```
1  Def Three-Stage Pipeline:
2  For [tileID, headID] in taskTable(CoreID):
3    qAddress=tokenTable(tileID, headID)
4    #QK on AIC's Cube Unit
5    Load Q using qAddress; Load K via blockTable;
6    QK = Q × K^T
7    #Softmax on AIV's Vector Unit
8    S = Softmax(QK)
9    #SV on AIC's Cube Unit
10   Load V via blockTable
11   O = S × V
12
13 Def Two-Stage Pipeline:
14 For [tileID, headID] in taskTable(blockIdx):
15   qAddress=tokenTable(tileID, headID)
16   #QK and Softmax on AIV's Vector Unit
17   Load Q using qAddress; Load K via blockTable;
18   QK = Q × K^T ;  S = Softmax(QK)
19   #SV on AIC's Cube Unit
20   Load V via blockTable
21   O = S × V
```

```
24 Def Four-Stage Pipeline:
25 For [tileID, headID] in taskTable(CoreID):
26   qAddress=tokenTable(tileID, headID)
27   Load Q using qAddress
28   Segment K/V into N blocks
29   For j in range(N)
30     #QK on AIC's Cube Unit
31     Load K_j via blockTable
32     QK_j=Q×K_j^T
33     #Softmax on AIV's Vector Unit
34     m_j=max(m_j-1, QK_j); S_j=exp(QK_j-m_j)
35     α_j=exp(m_j-1-m_j); l=α_j×l+rowsum(S_j)
36     #SV on AIC's Cube Unit
37     Load V_j via blockTable
38     O_j=S_j×V_j
39     #Update on AIV's Vector Unit
40     O=diag(α_j)^-1×O + O_j
41     if j == N-1:
42       O=diag(l)^-1×O
43 Operations on AIC's Cube Unit   Operations on AIV's Vector Unit
44
```

Figure 9: The Algorithm of Cube-Vector Orchestration.

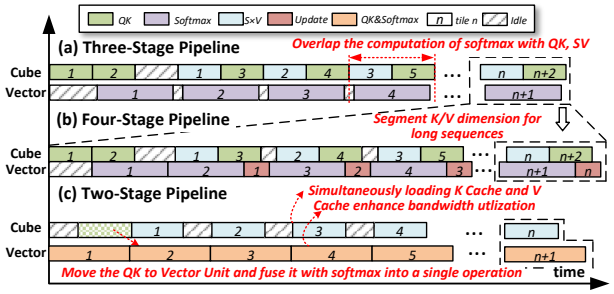

Figure 10: Pipeline of Meta-Attention.

simultaneously starting the QK computation for the second tiling data. This overlapping ensures that cube and vector units work concurrently, maximizing hardware utilization.

The intermediate data size for each tile is $tileSize \times kvLen$. When processing $coreNum$ tiles simultaneously, the total intermediate buffer size required is $pipeDepth \times tileSize \times kvLen \times coreNum$. By ensuring this buffer fits within the L2 cache, we can avoid accessing HBM. If the sequence length is short and the intermediate results fit into the L2 cache, there is no need to split along the K/V dimension. In such cases, the three-stage pipeline can be employed, avoiding additional computation and updates required for sequence splitting.

For extremely long sequences, however, splitting along the K/V dimension becomes necessary due to L2 cache limitations. This introduces an additional computation stage, where the $Softmax$ operation is divided into two steps: $Softmax$ and $Update$. The pipeline expands to four stages: $QK \rightarrow Softmax \rightarrow SV \rightarrow Update$. As shown in Fig. 10(b), we initially prefetch the QK and $Softmax$ computations for one tile of data. Subsequently, cube scheduling alternates between QK and SV tasks, while vector scheduling alternates between $Softmax$ and $Update$ tasks. This design enables parallel operation of the cube and vector units, maximizing hardware resource utilization and ensuring optimal performance for both short and long input sequences.

For fully decode-based tasks, we can adopt a new pipeline

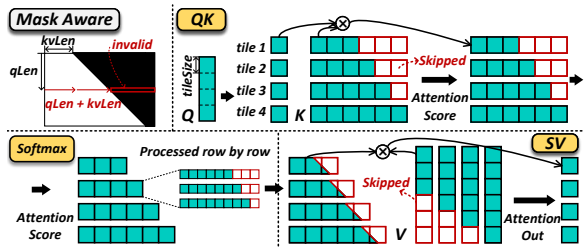

Figure 11: Mask-aware Computation.

design to optimize performance further and address the memory-bound issue of decode. Since the Query in decode stage consists of only a single token, the *Query* matrix is reduced to a vector. Consequently, the *QK* and *SV* computations transition from general matrix operations to matrix-vector operations. Executing these operations on cube unit would lead to inefficient utilization of resources. To address this, we move the *QK* operation to the vector unit and fuse the *QK* and $Softmax$ operations into a single operator. This fused operator ensures that the execution time for the vector unit aligns closely with the time required for the cube unit to perform the *SV* operation, effectively balancing the workload.

As illustrated in Fig. 10(c), the pipeline diagram shows that while the vector unit performs the *QK* and $Softmax$ operations for tile *n*, the cube unit concurrently executes the *SV* operation for *tile n-1*. Furthermore, the cube and vector units can simultaneously access the K/V cache data stored in HBM, improving memory bandwidth utilization and further boosting overall performance.

4.2.2 Exploiting Mask Sparsity

To fully utilize the sparsity in the attention mask and skip redundant computations, we adopt a mask-aware strategy that significantly enhances efficiency. As displayed in Fig. 11, in the attention computation, once the query dimension coordinate index *qLen* and the *kvLen* of K/V are known, any data corresponding to positions after $qLen + kvLen$ in each row of the attention score matrix is invalid. This insight allows us to guide the *QK*, $Softmax$, and *SV* operations to skip computations in these invalid regions.

In the *QK* operation, this principle enables us to directly omit the computation of corresponding results, effectively skipping the red tiling blocks in the result matrix. For the $Softmax$ operation, since it is calculated row by row, we can precisely control which attention scores are included in the computation at the token granularity. This token-level control enables us to support masks of arbitrary shapes while maintaining a mask-free design that eliminates the overhead of users generating and passing in masks externally. For the *SV* operation, the skipped computations occur during the reduced sum along the *K* dimension, where certain tiling blocks can be excluded based on the sparsity pattern. By applying these

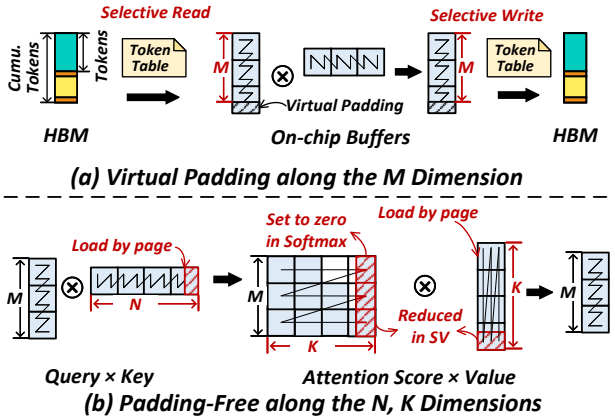

Figure 12: Handling Arbitrary Shapes of GEMM Operations.

optimizations across the *QK*, $Softmax$, and *SV* stages, we effectively exploit mask sparsity, ensuring highly efficient and flexible attention mechanisms.

5 SmoothGEMM

As discussed earlier, designing a matrix multiplication operation that supports arbitrary shapes while maintaining high performance across all possible shapes is a significant challenge. To address this, we adopt a memory-compute co-design strategy. Instead of optimizing matrix multiplication for every possible shape, we focus on maximizing performance for fixed shapes. To handle arbitrary shapes effectively, we introduce virtual padding at the on-chip memory level, along with selective read and write mechanisms. This approach allows matrix computations to accommodate a wide range of shapes seamlessly while still benefiting from the performance advantages of fixed-shape optimizations.

5.1 Virtual Padding on the M Dimension

As illustrated in Fig. 12(a), the dimension *M* in GEMM is intrinsically related to the number of tokens. In the case of Linear GEMM, the *M* dimension corresponds to the cumulative token count across P/D/V stages, while in Attention GEMM, it is determined by the token count in each stage. In cube-based or tensor-based AI accelerators, matrix computations are typically constrained by a minimum tiling size (e.g., 16×16 for cube cores). A common practice is to pad the *M* dimension to align with a multiple of the tiling size. However, this dynamic padding introduces non-trivial memory overhead and degrades performance.

To mitigate these issues, we replace the physical padding in global memory with virtual padding on the chip, combined with the selective read and write mechanisms shown in Fig. 12(a). This approach allows for efficient handling of matrices with arbitrary shapes. Specifically, on-chip buffer allocations

are made in tiling-size units to fully exploit the hardware's computational potential. During data transfer from global memory to the on-chip buffer, selective read operations copy only the actual, non-padding data. Similarly, selective write operations ensure that only the non-padding outputs are written back to global memory.

Because the virtual padding regions do not interfere with the computational results of the non-padding regions, this approach guarantees the correctness of the matrix computation results. Moreover, by limiting computations to fixed-shape matrix multiplications, we can apply highly customized optimizations for these fixed shapes, achieving both high efficiency and flexibility for dynamic workloads.

5.2 Optimizations for N and K Dimensions

As illustrated in Fig. 4(a), the dimensions of N and K in linear operations are determined by the model structure. These dimensions are typically multiples of hardware tile size, such as 16, eliminating the need for additional padding.

For Attention GEMM ops, dimensions N and K are tied to the sequence length, which can take arbitrary values. In theory, padding would be required for these dimensions. However, this is naturally handled by the K/V cache's block page structure, which stores and reads data in blocks aligned to multiples of tile size. As shown in Fig. 12(b), by reading entire blocks, the read length inherently conforms to the required hardware tile size. Furthermore, this padding does not affect the final Attention computation because we explicitly set the values in the padded regions to zero during $Softmax$ calculation. This ensures that the padded values are effectively excluded from the Attention score. Additionally, since the padded data is reduced during the Attention $score \times value$ operation, it does not influence the shape of the final output.

Therefore, matrix multiplication for arbitrary shapes can be efficiently supported without introducing additional padding overhead for the N and K dimensions.

5.3 Handling Dynamic Shapes

Given our focus on fixed shapes, such as multiples of the tiling size, and operating under budget constraints, we narrow the range of token sizes we handle. While the token count can vary significantly across different workloads, we apply a smoothing technique that reduces the shapes involved to fixed, well-defined sizes. This enables us to perform offline customization and optimization for these specific shapes, particularly for optimizations that are difficult to implement statically, such as swizzling [15].

Swizzling is a technique that optimizes matrix multiplication performance by altering the task allocation order across AI cores, enhancing L2 cache hit rates. However, determining the optimal allocation strategy for swizzling is complex and cannot be easily derived through theoretical formulas.

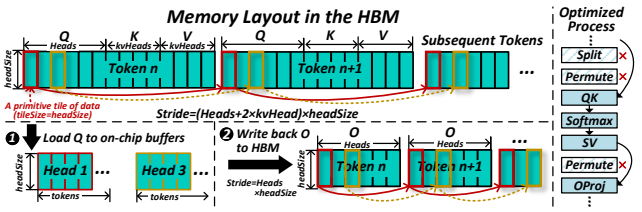

Figure 13: Elimination of Memory Operations.

To address this challenge, we conduct offline profiling to explore possible access patterns and identify the most efficient inter-core distribution strategy.

These optimal configurations are stored for future use. During online execution, we translate the dynamic workload into the nearest supported fixed shape. Using this fixed shape, we retrieve the pre-optimized configuration from the offline profiling results. This approach ensures that the dynamic workload is handled with the best configuration, enabling optimal performance during execution. By leveraging static optimizations, we can fully exploit the hardware's potential, even in the face of dynamic workloads.

5.4 Removing Memory Overheads

For the QKV input, its shape is $[tokens, heads + 2 \times kvHeads, headSize]$, where the outermost dimension corresponds to the tokens, and each token contains its QKV fusion. The attention operation is performed in parallel along the $Head$ dimension, which requires the $Head$ dimension to be positioned as the outermost dimension. Moreover, we need to read the Q, K, and V separately rather than the fused QKV tensor. Typically, $Split$ and $Permute$ operations are introduced to reorient the tensor for efficient computation. After the attention computation, another $Permute$ operation is applied to transform the output O back to the shape $[tokens, heads, headSize]$.

To reduce memory overhead, we fuse $Split$ and $Permute$ directly into GEMM computations shown in Fig. 13. For the QK operation, tile-based and stride-based reads are employed to access the required data directly from the QKV tensor, eliminating the need for separate $Split$ and $Permute$ steps. Similarly, for the SV computation, tile-based and stride-based writes are used to store the results in a format that is directly aligned with the $OProj$. This integration removes explicit $Permute$ operations, allowing the SV output to seamlessly flow into subsequent matrix multiplication operations.

6 Evaluation

6.1 XY-Serve Implementation

We built an Ascend-native inference system based on vLLM [31], leveraging Ascend intrinsic to implement core mod-

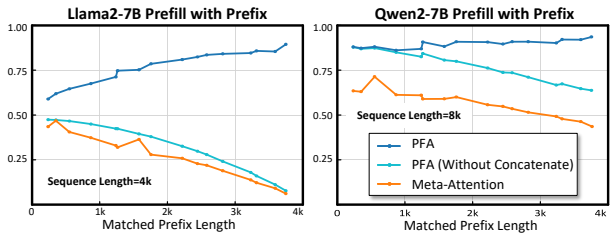

Figure 14: Performance of Prefill Attention with Prefix.

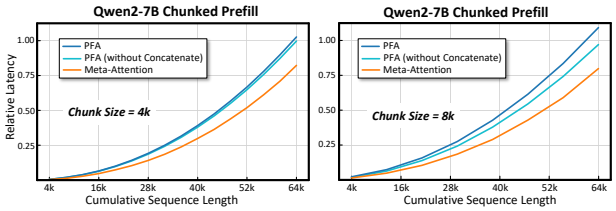

Figure 15: Long Sequence Attention with Chunked Prefill.

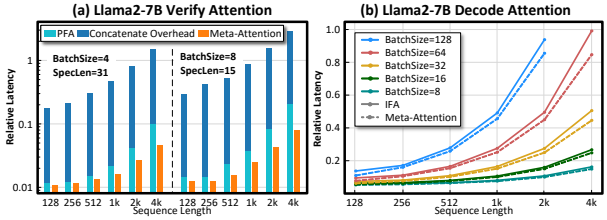

Figure 16: Performance of LLM Verify and Decode Attention.

ules such as SmoothGEMM, Meta-Attention, and other essential operators like normalization, activation, and embedding. These operators were exposed to the Python API via pybind11 [7] and seamlessly replaced the corresponding GPU kernels in vLLM, enabling vLLM to support the Ascend NPU.

To reduce the overhead of frequent Python calls, we offloaded the entire model-forward process to C++, integrating the optimized operators into a single C++ function. This model-forward function is then exposed to vLLM for invocation, ensuring a streamlined and efficient execution path.

Additionally, we replaced the native vLLM scheduler with a token-wise scheduling strategy, integrating workload decomposition and computation task reordering to better support dynamic workloads. We redesigned the speculative decoding framework in vLLM, enabling token tree construction and metadata generation for Meta-Attention. These enhancements allow us to implement tree-based speculative decoding algorithms, such as Lookahead Decoding [46], further improving inference performance.

6.2 Performance of Meta-Attention

In this section, we evaluate the performance of the attention kernel under dynamic workloads typically encountered in real-world systems. The comparison targets are PromptFlashAttention (PFA) [12] and IncreFlashAttention (IFA) [11] from torch-npu 2.1 [6].

6.2.1 Prefill Attention with Arbitrary Prefix

In practical systems, the length of the matched system prefix can vary arbitrarily. Therefore, it is crucial to assess performance under arbitrary-length prefix reuse. To simulate this behavior, we adjust the number of reused tokens for an input prompt, token by token, and evaluate performance under different lengths of system prefix matched. Fig. 14 shows the performance under different system prefix reuse (ranging from 0 to $4k$) for $4k$ and $8k$ prompt inputs. The results show that as the system prefix increases, the processing time of our meta-attention kernel decreases. However, the PFA kernel does not benefit from prefix reuse, primarily because its prefill kernel does not support PagedAttention and only accepts continuous Q, K, and V. When we concatenate the prefix hits from the K/V cache with the new K and V, the concatenation time increases as the reuse length grows, counteracting the benefit of prefix reuse. Even when comparing the computation time of the PFA kernel (excluding the K/V concatenation process), our kernel performs an average of 22.4% better.

6.2.2 Chunked Prefill with Long Sequences

For processing long sequences, chunking the sequence into smaller segments is a widely used approach. On the one hand, chunking allows for sequence parallelization by combining it with pipeline parallelism [16, 39]. On the other hand, it reduces the impact of prefill on decoding interruptions [17]. The Chunked Prefill method splits long sequences into multiple chunks, processing them sequentially. After processing each chunk, the corresponding keys and values are stored in the K/V cache for reuse by subsequent chunks. Fig. 15 shows the performance of our Chunked Prefill method for long sequences (with chunk sizes set to $4k$ and $8k$ and a sequence length of up to $64k$). The results demonstrate that our performance surpasses PFA across all sequence lengths. Even when comparing pure computation time with PFA, our kernel shows an improvement up to 22.2%.

6.2.3 Speculative Decoding

Next, we evaluate the performance of the verify kernel under different context lengths. We compare the performance with a $batchSize = 4$, $specLen = 31$ and a $batchSize = 8$, $specLen = 15$. Fig. 16(a) shows that, across different context lengths, our kernel consistently outperforms PFA. Furthermore, as the context length increases, the performance improvement becomes increasingly time-consuming as the historical length grows. Even when excluding this concatenation operation

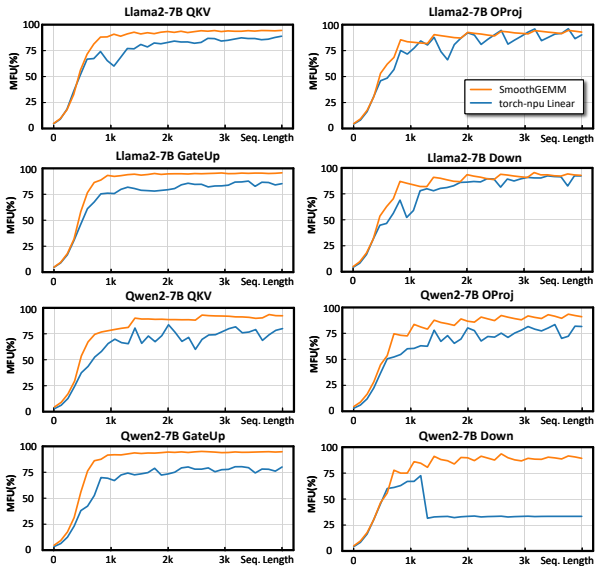

Figure 17: Linear GEMM Performance.

from PFA and comparing only the pure computation time, our kernel still demonstrates an average improvement of 28.6%.

6.2.4 Decode Performance

In the decode phase, both the context length and batch size can vary arbitrarily. To flexibly support decoding with arbitrary context lengths, we enable PagedAttention optimization. To evaluate decoding performance under such conditions, we measure performance across different context lengths and batch sizes. Fig. 16 presents the performance of Llama2-7B under varying sequence lengths and batch sizes. The results show that, compared to the IFA kernel, our kernel achieves performance improvements across all combinations of batch size and sequence length, with average 12.9% improvements.

6.3 Performance of SmoothGEMM

In practical systems, the input length from users can vary arbitrarily, ranging from 0 to the maximum length supported by the model. For long sequences, to minimize the impact of prefill on decode and to optimize sequence parallelization, we typically adopt the Chunked Prefill strategy, which imposes a constraint on the maximum chunk length, such as $4k$. In real-world scenarios, lengths smaller than the chunk size may also be encountered. To evaluate performance across different conditions, we assess the LLMs with input lengths ranging from 1 to the chunk size (4096).

We compare the performance of linear operators (*QKV*, *OProj*, *GateUp*, and *Down*) using shapes derived from Llama2-7B and Qwen2-7B with TP=1. As displayed in Fig. 17, the results indicate that SmoothGEMM outperforms torch-npu linear by an average of 14.6%, demonstrating superior

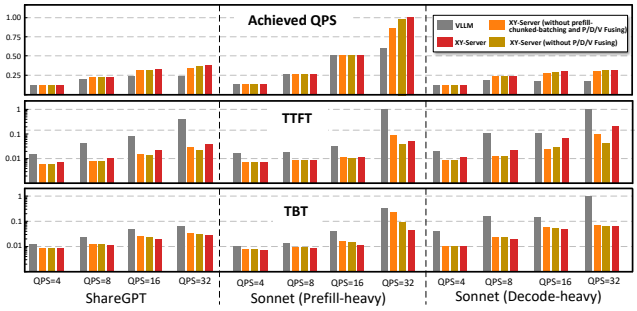

Figure 18: End-to-End Evaluation on Nightly Benchmarks.

performance across nearly all tested shapes. Moreover, the performance remains stable, with ideal MFU typically achieved for sequence sizes above $1k$.

6.4 End-to-End Evaluation

6.4.1 vLLM Nightly Benchmarks

For end-to-end benchmarking, we use the nightly-benchmarks from the vLLM community [13], with Qwen2-7B as the model. The baseline comparison is against a community version of vLLM that supports Ascend NPU (Ascend-vLLM) [14], primarily utilizing GEMM provided by torch-npu and Fused-Attention operators for computation. The test data is divided into three scenarios: ShareGPT, Prefill-heavy(462 input tokens and 16 output tokens), and Decode-heavy(462 input tokens and 256 output tokens).

As shown in Fig. 18, we measure performance under fixed request rates per second (QPS) of 4, 8, 16, and 32 for each test dataset. The following metrics are collected: average Time-to-First-Token (TTFT), average Time-between-Tokens (TBT), and achieved QPS. Even without enabling advanced features such as prefill-chunked-batching and P/D/V fusing, XY-Serve demonstrates a clear performance advantage over the baseline. Specifically, XY-Serve achieves an achieved QPS improvement of up to 79% across various workload types. Additionally, it delivers 64% lower average TTFT and 57% lower average TBT latency. This improvement is primarily attributed to the efficient optimization of operator implementations.

With the dynamic scheduling optimizations of prefill-chunked-batching and PDV fusing enabled, XY-Serve further gains an achieved QPS improvement up to 89% and reduces average TBT latency by 69% across all scenarios. This outcome underscores XY-Serve's strong support for dynamic workloads, effectively benefiting from these enhancements.

When prefill-chunked-batching is enabled, the length of tokens processed in each prefill is effectively maintained at the budgeted length, improving MFU and reducing TTFT under high-pressure conditions. However, enabling P/D/V fusing results in a slight deterioration of TTFT latency in the Decode-heavy scenario under high throughput pressure. In this case,

the number of decodes fused with the prefill increases, which slightly impacts the TTFT.

6.4.2 Ascend NPUs VS. GPUs

We compared the end-to-end inference MFU and MBU between XY-Serve running on the 910B and the official vLLM-v0.6.4.post1 on the Nvidia A800. The measurements were taken during the prefill and decode stages of the entire forward pass for the Qwen2-7B and Llama2-7B models at TP=1, across various sequence lengths. As shown in Fig. 19, XY-Serve achieves MFU and MBU similar to the A800. Notably, in terms of MBU, XY-Serve demonstrates a clear advantage over GPUs, showing an improvement up to 17%.

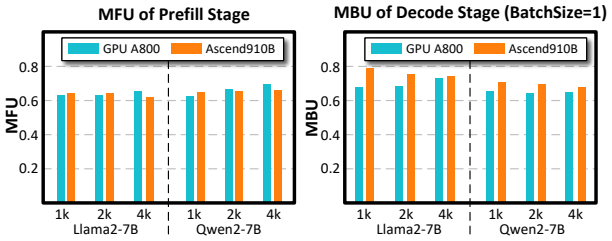

Figure 19: Comparison between Ascend NPUs and GPUs.

7 Related Work

Attention Optimization: FlashAttention1 [24] and FlashAttention2 [23] optimize the prefill phase by tiling computations to avoid HBM access, improving performance. FlashAttention3 [40] further enhances performance through parallelism between softmax and matrix operations. FastAttention [36] extends FlashAttention2 from GPUs to Ascend NPUs, while FlashDecoding [5] improves decoding efficiency for small batches by splitting along the sequence dimension. Recent works [2,43] further optimize decoding performance by transforming GEMV operations into GEMM operations when sharing prefixes. While these techniques primarily target either the prefill or decode phases, POD-Attention [29] simultaneously optimizes both, maximizing computational power and bandwidth. In contrast, our work tackles real-world deployment scenarios with dynamic prefix reuse and speculative algorithms, leading to mixed P/D/V stages. We decompose dynamic workloads into hardware-friendly meta-primitives, simplifying attention module design.

Linear Optimization: Existing techniques like swizzling [15], split-k [1], and ping-pong [3] are commonly used in linear optimization. Our approach shows that supporting specific matrix shapes is sufficient for dynamic LLM workloads. By optimizing these shapes offline using the above techniques, we store the configurations and apply them during online execution to achieve optimal performance.

Serving Systems: Several works aim to address the dynamic nature of inference systems. Orca [44] mitigates output length variability through iteration-level scheduling, while PageAttention [31] optimizes memory allocation for KV caches, reducing waste from fixed-length allocations. Our approach builds on these, tackling additional complexities in real-world inference systems, such as hybrid P/D/V stages and dynamic shapes in each stage.

The interruption of prefill on decode can increase TBT. Two strategies have been proposed to address this: SplitFuse [17,18,26], which divides the Prefill phase into smaller chunks and fuses chunks with decode and disaggregated LLMs [27, 28, 39, 48], which separate Prefill and Decode across different machines. XY-Serve supports both two deployments. It also enables seamless transitions between Prefill and Decode roles in disaggregated setups.

8 Conclusion

We introduced XY-Serve, an end-to-end production serving system designed to tackle challenges of dynamic LLM workloads. By integrating workload decomposition, computation task reordering, and meta kernels (Meta-Attention and SmoothGEMM), XY-Serve optimizes throughput, latency, and computational efficiency in real-world production environments. Experimental results demonstrate superior performance on Ascend NPUs, delivering MFU and MBU comparable to GPUs. With its flexibility to handle diverse dynamic workloads, XY-Serve sets a new benchmark for efficiency and adaptability in production-grade LLM inference systems.

References

[1] Accelerating Llama3 FP8 Inference with Triton Kernels. https://pytorch.org/blog/accelerating-llama3/?hss_channel=lcp-78618366/.

[2] Accelerating Self-Attentions for LLM Serving with FlashInfer | FlashInfer. https://flashinfer.ai/2024/02/02/introduce-flashinfer.html.

[3] Deep Dive on CUTLASS Ping-Pong GEMM Kernel | PyTorch. https://pytorch.org/blog/cutlass-ping-pong-gemm-kernel/.

[4] Faster Text Generation with Self-Speculative Decoding. https://huggingface.co/blog/layerskip.

[5] Flash-Decoding for long-context inference. https://pytorch.org/blog/flash-decoding/.

[6] GitHub - Ascend/pytorch: Ascend PyTorch adapter (torch_npu). Mirror of https://gitee.com/ascend/pytorch. https://github.com/Ascend/pytorch.

[7] GitHub - pybind/pybind11: Seamless operability between C++11 and Python. https://github.com/pybind/pybind11.

[8] Introduction — vLLM. https://docs.vllm.ai/en/latest/automatic_prefix_caching/apc.html.

[9] Torch.nn-Native PyTorch APIs-PyTorch2.1-API List-PyTorch Network Model Porting and Training Guide-Model development (PyTorch)-7.0.0-CANN commercial edition-Ascend Documentation-Ascend Community. https://www.hiascend.com/document/detail/en/canncommercial/700/modeldevpt/ptmigr/ptaoplist_000006.html.

[10] Torch_npu.npu_fused_infer_attention_score. https://www.hiascend.com/doc_center/source/zh/Pytorch/60RC2/apiref/apilist/ptaoplist_000787.html.

[11] Torch_npu.npu_incre_flash_attention. https://www.hiascend.com/doc_center/source/zh/Pytorch/60RC2/apiref/apilist/ptaoplist_000788.html.

[12] Torch_npu.npu_prompt_flash_attention. https://www.hiascend.com/doc_center/source/zh/CANNCommunityEdition/80RC1alpha001/apiref/fmkadptapi/ptaoplist_000142.html.

[13] vllm nightly-benchmarks. https://github.com/vllm-project/vllm/tree/main/.buildkite/nightly-benchmarks.

[14] vllm support for ascend npu. https://github.com/vllm-project/vllm/pull/8054.

[15] Optimizing Compute Shaders for L2 Locality using Thread-Group ID Swizzling. https://developer.nvidia.com/blog/optimizing-compute-shaders-for-l2-locality-using-thread-group-id-swizzling/, July 2020.

[16] Amey Agrawal, Junda Chen, Íñigo Goiri, Ramachandran Ramjee, Chaojie Zhang, Alexey Tumanov, and Esha Choukse. Mnemosyne: Parallelization strategies for efficiently serving multi-million context length llm inference requests without approximations. *arXiv preprint arXiv:2409.17264*, 2024.

[17] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in llm inference with sarathi-serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 117–134, 2024.

[18] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369*, 2023.

[19] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[20] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D Lee, Deming Chen, and Tri Dao. Medusa: Simple llm inference acceleration framework with multiple decoding heads, 2024. *URL https://arxiv.org/abs/2401.10774*.

[21] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-end object detection with transformers. In *European conference on computer vision*, pages 213–229. Springer, 2020.

[22] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023.

[23] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning (2023). *arXiv preprint arXiv:2307.08691*, 2023.

[24] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

[25] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2021.

[26] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, et al. Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference. *arXiv preprint arXiv:2401.08671*, 2024.

[27] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, et al. Inference without interference: Disaggregate llm inference for mixed downstream workloads. *arXiv preprint arXiv:2401.11181*, 2024.

[28] Yibo Jin, Tao Wang, Huimin Lin, Mingyang Song, Peiyang Li, Yipeng Ma, Yicheng Shan, Zhengfan Yuan, Cailong Li, Yajing Sun, et al. P/d-serve: Serving disaggregated large language model at scale. *arXiv preprint arXiv:2408.08147*, 2024.

[29] Aditya K Kamath, Ramya Prabhu, Jayashree Mohan, Simon Peter, Ramachandran Ramjee, and Ashish Panwar. Pod-attention: Unlocking full prefill-decode overlap for faster llm inference. *arXiv preprint arXiv:2410.18038*, 2024.

[30] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[31] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

[32] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pages 19274–19286. PMLR, 2023.

[33] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. Eagle: Speculative sampling requires rethinking feature uncertainty. *arXiv preprint arXiv:2401.15077*, 2024.

[34] Heng Liao, Jiajin Tu, Jing Xia, Hu Liu, Xiping Zhou, Honghui Yuan, and Yuxing Hu. Ascend: a scalable and unified architecture for ubiquitous deep neural network computing: Industry track paper. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 789–801. IEEE, 2021.

[35] Heng Liao, Jiajin Tu, Jing Xia, and Xiping Zhou. Davinci: A scalable architecture for neural network computing. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–44. IEEE Computer Society, 2019.

[36] Haoran Lin, Xianzhi Yu, Kang Zhao, Lu Hou, Zongyuan Zhan, Stanislav Kamenev, Han Bao, Ting Hu, Mingkai Wang, Qixin Chang, et al. Fastattention: Extend flashattention2 to npus and low-resource gpus. *arXiv preprint arXiv:2410.16663*, 2024.

[37] Xiaoxuan Liu, Cade Daniel, Langxiang Hu, Woosuk Kwon, Zhuohan Li, Xiangxi Mo, Alvin Cheung, Zhijie Deng, Ion Stoica, and Hao Zhang. Optimizing speculative decoding for serving large language models using goodput. *arXiv preprint arXiv:2406.14066*, 2024.

[38] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, et al. Specinfer: Accelerating large language model serving with tree-based speculative inference and verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 932–949, 2024.

[39] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: A kvcache-centric disaggregated architecture for llm serving. *arXiv preprint arXiv:2407.00079*, 2024.

[40] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *arXiv preprint arXiv:2407.08608*, 2024.

[41] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[42] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, et al. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*, 2024.

[43] Lu Ye, Ze Tao, Yong Huang, and Yang Li. ChunkAttention: Efficient self-attention with prefix-aware KV cache and two-phase partition. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 11608–11620, Bangkok, Thailand, August 2024. Association for Computational Linguistics.

[44] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for transformer-based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022.

[45] Jun Zhang, Jue Wang, Huan Li, Lidan Shou, Ke Chen, Gang Chen, and Sharad Mehrotra. Draft & verify: Lossless large language model acceleration via self-speculative decoding. *arXiv preprint arXiv:2309.08168*, 2023.

[46] Yao Zhao, Zhitian Xie, Chen Liang, Chenyi Zhuang, and Jinjie Gu. Lookahead: An inference acceleration framework for large language model with lossless generation accuracy. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 6344–6355, 2024.

[47] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs, 2024. *URL https://arxiv.org/abs/2312.07104*.

[48] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, 2024.