CoDec: Prefix-Shared Decoding Kernel for LLMs

ZHIBIN WANG*, State Key Laboratory for Novel Software Technology, Nanjing University, China
RUI NING*, State Key Laboratory for Novel Software Technology, Nanjing University, China
CHAO FANG, State Key Laboratory for Novel Software Technology, Nanjing University, China
ZHONGHUI ZHANG, State Key Laboratory for Novel Software Technology, Nanjing University, China
XI LIN, State Key Laboratory for Novel Software Technology, Nanjing University, China
SHAOBO MA, State Key Laboratory for Novel Software Technology, Nanjing University, China
MO ZHOU, State Key Laboratory for Novel Software Technology, Nanjing University, China
XUE LI, Alibaba Group, China
ZHONGFENG WANG, State Key Laboratory for Novel Software Technology, Nanjing University, China
CHENGYING HUAN, State Key Laboratory for Novel Software Technology, Nanjing University, China
RONG GU†, State Key Laboratory for Novel Software Technology, Nanjing University, China
KUN YANG, State Key Laboratory for Novel Software Technology, Nanjing University, China
GUIHAI CHEN, State Key Laboratory for Novel Software Technology, Nanjing University, China
SHENG ZHONG, State Key Laboratory for Novel Software Technology, Nanjing University, China
CHEN TIAN, State Key Laboratory for Novel Software Technology, Nanjing University, China

Prefix-sharing among multiple prompts presents opportunities to combine the operations of the shared prefix, while attention computation in the decode stage, which becomes a critical bottleneck with increasing context lengths, is a memory-intensive process requiring heavy memory access on the key-value (KV) cache of the prefixes. Therefore, in this paper, we explore the potential of prefix-sharing in the attention computation of the decode stage. However, the tree structure of the prefix-sharing mechanism presents significant challenges for attention computation in efficiently processing shared KV cache access patterns while managing complex dependencies and balancing irregular workloads. To address the above challenges, we propose a dedicated attention kernel to combine the memory access of shared prefixes in the decoding stage, namely CoDec. CoDec delivers two key innovations: a novel shared-prefix attention kernel that optimizes memory hierarchy and

*Both authors contributed equally to this research.
†Corresponding author.

Authors' Contact Information: Zhibin Wang, wzbwangzhibin@gmail.com, State Key Laboratory for Novel Software Technology, Nanjing University, China; Rui Ning, rning@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, China; Chao Fang, chao.fang@kuleuven.be, State Key Laboratory for Novel Software Technology, Nanjing University, China; Zhonghui Zhang, zhonghuizhang@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, China; Xi Lin, 350904583lx@gmail.com, State Key Laboratory for Novel Software Technology, Nanjing University, China; Shaobo Ma, shaoboma@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, China; Mo Zhou, 221900277@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, China; Xue Li, youli.lx@alibaba-inc.com, Alibaba Group, China; Zhongfeng Wang, fantasysee@foxmail.com, State Key Laboratory for Novel Software Technology, Nanjing University, China; Chengying Huan, huanzhizun888@126.com, State Key Laboratory for Novel Software Technology, Nanjing University, China; Rong Gu, gurong@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, China; Kun Yang, kunyang@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, China; Guihai Chen, gchen@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, China; Sheng Zhong, sheng.zhong@gmail.com, State Key Laboratory for Novel Software Technology, Nanjing University, China; Chen Tian, tianchen@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, China.

ACM 2836-6573/2026/6-ART151
https://doi.org/10.1145/3802028

exploits both intra-block and inter-block parallelism, and a comprehensive workload balancing mechanism that efficiently estimates cost, divides tasks, and schedules execution. Experimental results show that CoDec achieves an average 1.9× speedup and 120.9× memory access reduction compared to the state-of-the-art FlashDecoding kernel regarding attention computation in the decode stage and 3.8× end-to-end time per output token compared to the vLLM.

CCS Concepts: • **Computing methodologies** → **Machine learning**; **Parallel algorithms**; • **Information systems** → **Data management systems**.

Additional Key Words and Phrases: GPU kernel, large language models (LLMs), attention computation, prefix-sharing

ACM Reference Format:
Zhibin Wang, Rui Ning, Chao Fang, Zhonghui Zhang, Xi Lin, Shaobo Ma, Mo Zhou, Xue Li, Zhongfeng Wang, Chengying Huan, Rong Gu, Kun Yang, Guihai Chen, Sheng Zhong, and Chen Tian. 2026. CoDec: Prefix-Shared Decoding Kernel for LLMs. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 151 (June 2026), 27 pages. https://doi.org/10.1145/3802028

Code Availability: Code is available at https://github.com/wzbxpy/codec.
Earlier Version: The earlier arXiv version of this paper was titled Flashforge.

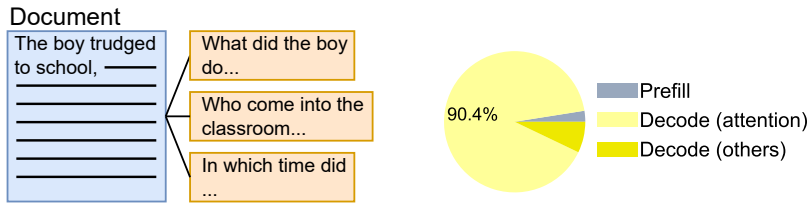
1 Introduction

Large language models (LLMs) have demonstrated significant performance across diverse tasks, such as question answering [21], planning [40, 41], code generation [15, 37], recommendation systems [19, 59], and even solving complex mathematical problems [9, 32, 35, 50]. Despite their impressive capabilities, inference efficiency remains critical for LLM deployment, as it directly impacts user experience and operational costs [13, 63]. For instance, real-time applications such as chatbots and interactive AI agents require low-latency responses to maintain seamless interactions [27]. Meanwhile, enterprise-scale deployments, where LLMs process millions of queries daily, must optimize computational resources to remain cost-effective [8, 22, 28].

One of the most promising directions for improving LLM inference efficiency is prefix-sharing founded on the observation that many prompts share identical prefixes, which is common in document question answering [12, 23, 51], tree-of-thoughts [55], speculative decoding [29], and few-shot prompting [36]. For example, as shown in Figure 1 (a), in document-based question answering, multiple questions may pertain to the same document [12, 23, 51]. For prefill stage generating the KV cache for the prompt tokens, prefix-sharing can be utilized to reduce the duplicated computation and memory consumption of the KV cache generation of the shared prefix across different requests [34, 42, 45, 47, 58, 60, 61].

With increasing context lengths, the decoding stage, particularly attention computation, becomes the critical bottleneck in LLM inference, accounting for 90% of total time as shown in Figure 1 (b) when running 100K prompts with 128 output tokens on Llama-3.1-8B [2]. Different from the prefill stage generating the KV cache, the decode stage autoregressively generates the output tokens based on the generated KV cache, thereby sequentially processing the tokens one by one and requiring heavy memory access to the KV cache, exhibiting insufficient parallelism and memory-bound pattern [14]. Recently, state-of-the-art attention optimization methods, such as FlashAttention [7] and FlashDecoding [8], have achieved significant speedups by leveraging shared memory (i.e., on-chip memory) to reduce memory access overhead and by increasing parallelism in attention computation. Consider a batch of multi-head attention that takes three 4D tensors as input: the query ($\mathbf{Q} \in \mathbb{R}^{bs \times n_q \times h \times d}$), key ($\mathbf{K} \in \mathbb{R}^{bs \times n \times h \times d}$), and value ($\mathbf{V} \in \mathbb{R}^{bs \times n \times h \times d}$) tensors[1]. FlashAttention

[1]The notation of the tensors is shown in Table 1.



(a) Prefix-shared document QA. (b) Cost of various operations.

Fig. 1. Motivation of leveraging prefix-sharing in attention of decoding stage.

and FlashDecoding decompose the attention operation in the *batch, head, query sequence, and KV sequence dimensions* into multiple blocks so that each block fits in shared memory and exposes four-way parallelism. However, when employing these techniques in prefix-sharing scenarios, queries sharing identical prefixes are processed individually by separate computational units, even though they could potentially share memory access for the common KV cache. This processing pattern inevitably results in duplicated memory transactions for accessing the shared KV cache.

From a data management perspective, the KV cache is a large, dynamic, and performance-critical data structure: serving systems must decide how to lay out, reuse, and access KV blocks efficiently under strict latency constraints. CoDec targets the decode stage where KV access becomes bandwidth-bound, and treats shared-prefix attention as a data-access optimization problem—reducing redundant reads to shared KV blocks by coalescing access across requests. This is complementary to higher-level KV cache management and scheduling work (e.g., HotPrefix [25] and Cache-Craft [3]), which focuses on what to keep and when to reuse; CoDec focuses on how to execute the shared-prefix accesses efficiently once such reuse opportunities exist.

In this paper, we explore leveraging prefix-sharing in the decode stage, specifically by optimizing the memory access patterns for shared KV cache across different requests. This approach directly addresses the redundant memory transactions identified above, targeting the primary performance bottleneck in LLM inference. However, when shifting the regular attention computation [7, 8] between 4D tensors to the irregular shared prefix attention computation, there are several challenges to be addressed:

Challenge 1: Organizing complex dependencies in shared prefix attention computation. The management of the shared prefix KV cache has been well studied in the prefill stage [54, 60], logically organized as a radix tree of 3D tensors where each node represents a chunk of the prefix KV cache. However, further extending this tree structure to attention computation introduces two major issues: First, it requires not only the KV cache but also the corresponding query tensor shared with the prefix to coordinate KV cache access, complicating management. Second, reduction operations are needed to combine decoding results in corresponding prefix KV cache nodes for each query in the tree structure, which is nontrivial to be parallelized. Prior art [56, 57] only considered the trivial case where all requests share the same prefix, thus simply handling shared and non-shared computations separately. Hence, efficiently leveraging the KV cache tree structure for both attention and reduction operations remains challenging yet essential.

Challenge 2: Balancing the workload of irregular prefix-shared attention computation. The workload for attention computation between each KV cache node and its corresponding query tensor is determined by both query count and prefix length of the KV cache node, which vary significantly across computations, leading to highly irregular workloads [30]. Moreover, varying shapes of query tensors and prefix KV cache nodes result in divergent memory or compute bounds [48, 62], making theoretical workload estimation impractical. This necessitates an integrated

solution combining cost estimation, intelligent task division, and efficient scheduling to balance workloads without resorting to expensive fine-grained partitioning.

To address the above challenges, we develop the dedicated prefix-shared attention operator, namely CoDec, which combines the memory access of the attention computation of the shared prefix across different requests in the decode stage. CoDec delivers two key innovations: First, it implements a novel shared-prefix attention kernel that optimizes both memory hierarchy between shared and global memory and exploits intra-block and inter-block parallelism. Second, it incorporates a comprehensive workload balancing mechanism with a cost estimator, task divider, and scheduler that guides execution before the attention kernel runs. It should be noted that CoDec follows the same paged KV-cache layout as PagedAttention and exposes an attention interface compatible with FlashDecoding, enabling straightforward integration into existing serving stacks such as vLLM. We summarize the contributions of this paper as follows:

Shared prefix attention kernel (Section 4). To efficiently form the attention computation, we introduce the indexes between the prefix KV cache tree and the query tensors, which facilitate loading the corresponding tensors to the shared memory. Moreover, we abstract two fundamental primitives in the block-level, i.e., *partial attention computation (PAC)* to compute the partial output between 2D query and KV tensors extracted from global query and KV tensors, and *partial output reduction (POR)* to reduce two partial outputs of the same query. Building on these two primitives, we propose an inter-block computation task executor and a dedicated tree-based reduction. The tree-based reduction aims to maximize the GPU utilization by achieving a parallelism degree equal to the block number while minimizing the number of reduction operations.

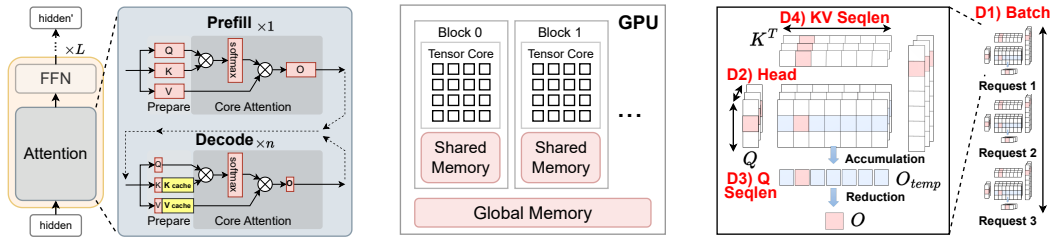
Workload balancing mechanism (Section 5). Directly mapping the partial attention computation between each KV cache node and the corresponding query tensor suffers from significant irregular workload and insufficient parallelism. Therefore, we further divide the computation into multiple subtasks. Recognizing that the workload of each subtask is neither determined by IO complexity nor compute complexity as shown in Table 2, we propose a profile-based cost estimator to guide the task division. Moreover, we formulate the optimization problem of task division and scheduling, unfortunately, it is NP-hard. However, given the specific characteristics of partial attention computation, i.e., coarse-grained has less overhead than fine-grained, we propose a greedy algorithm to solve the problem.

Evaluation (Section 7). We evaluate CoDec across a range of prefix-sharing workloads, GPUs, attention variants, and model sizes. Compared to FlashDecoding, CoDec achieves an average 1.9× speedup and a 120.9× reduction in global memory access for decode-stage attention, and achieves a 3.8× TPOT speedup over vLLM in end-to-end comparison.

2 Background

2.1 LLM Inference

Transformer architecture: Recent mainstream LLMs, such as ChatGPT [31], DeepSeek [11], Llama [17], and Gemini [43] are based on the transformer architecture [44], which generates tokens in an auto-regressive manner. As shown in Figure 2 (a), the transformer consists of the attention module and the feed-forward network (FFN) module. The attention module calculates the attention scores between each pair of tokens, allowing the model to learn the relationships and dependencies among them, which makes the transformer model outperform the RNN model [26] considering the long-range dependencies [44]. The FFN module is responsible for learning complex representations of the tokens. These two modules are typically stacked L times to deepen the model.



(a) Workflow of prefill and decode in Transformer. (b) GPU architecture. (c) Example of FlashAttention/FlashDecoding.

Fig. 2. Background knowledge of LLM inference, GPU architecture, and existing attention kernels.

Prefill and decode stages: The inference process of these models comprises two primary stages: prefill and decode, which are depicted in Figure 2 (a). During the prefill stage, the model simultaneously processes the input token sequence, caches the corresponding key and value (KV) tensors for these tokens, and generates the initial output token. In the subsequent decode stage, the model processes one token per step, generating the next token autoregressively based on previously generated tokens and the cached KV tensors. Because the prefill stage processes numerous tokens concurrently, it is computationally intensive and typically compute-bound. Conversely, the decode stage processes only a single token per step, resulting in significantly lower computational demands and rendering this stage memory-bound.

Figure 1 (b) presents the prefill/decode time breakdown of two workloads when serving Llama-3.1-8B. When running 100K prompts with 128 output tokens, the overall decoding latency is 102s, while the prefill latency is only 2.62s, and the attention kernel accounts for 90% of the total time. Furthermore, as sequence length increases, the proportion of time consumed by the attention kernel also rises.

In summary, with the increasing context length, *the attention computation in the decode stage becomes the critical bottleneck of the LLM inference process.* Therefore, it is crucial to optimize the attention computation in the decode stage.

Table 1. Notation Table.

Notation	Definition
Q, K, V, O	Partial 2D Query, key, value and output tensor
$\mathbf{Q}, \mathbf{K}, \mathbf{V}, \mathbf{O}$	Query, key, value and output tensor of batch
S, P	Attention score before and after softmax function
h, d	Number of attention heads, hidden layer dimension
bs, n, n_q	Batch size, sequence length, query tokens' length
T	Set of tasks
p, t	Number of parallel thread blocks and tasks
b_q, b_k	Number of slices in the query, KV cache dimension

2.2 Attention Mechanism

Given the significance of the attention mechanism in both algorithm-level and system-level, we briefly review the self-attention operation and its variants in execution.

Self-attention operation: The key idea of the self-attention operator is to compute the attention score between each pair of tokens i and j, which indicates the importance of token j in the context

of token i. Subsequently, the embedding of token i is updated by a weighted sum of the embeddings of all tokens, where the weights are determined by the attention scores.

Formally, given a sequence of input tokens, three linear transformations generate the query ($Q \in \mathbb{R}^{n_q \times d}$), key ($K \in \mathbb{R}^{n \times d}$), and value ($V \in \mathbb{R}^{n \times d}$) tensors, where n is the sequence length, d is the hidden size, and n_q is the sequence length of the query tokens. Note that in some cases $n_q \neq n$, such as in the decode stage or when leveraging chunked-prefill [4].

$$O = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V, \tag{1}$$

where $O \in \mathbb{R}^{n_q \times d}$ is the output tensor indicating the updated embedding of the query tokens. The softmax function is employed to normalize the attention scores of each query token, i.e., on each row. Given a vector $\mathbf{x} \in \mathbb{R}^n$, the softmax function is defined as:

$$m = \max(\mathbf{x}), s = \sum_{i=1}^{n} e^{\mathbf{x}[i]-m}, \text{softmax}(\mathbf{x})[i] = \frac{e^{\mathbf{x}[i]-m}}{s}. \tag{2}$$

m is the maximum value of the vector $\mathbf{x}$ to prevent overflow in the exponentiation operation [7].

In this paper, we use multi-head attention [44] as the default attention mechanism, and other attention mechanisms [5, 10, 38] will be discussed in Section 8. Another consideration is batching, which is a common technique used to improve resource utilization by enlarging the workload of processing multiple requests in parallel.

Overall, the general attention operation for batched multi-head attention takes three input tensors: the query ($\mathbf{Q} \in \mathbb{R}^{bs \times n_q \times h \times d}$), key ($\mathbf{K} \in \mathbb{R}^{bs \times n \times h \times d}$), and value ($\mathbf{V} \in \mathbb{R}^{bs \times n \times h \times d}$) tensors.

Attention variants in training, prefill, and decode stages:

- **Training** directly uses the general attention operation and $n_q = n$.
- **Prefill stage** usually sets the batch size $bs = 1$ as there is enough workload in the prefill stage.
- **Decode stage** only processes one token for each request at a time, i.e., $n_q = 1$, which *suffers from both insufficient parallelism and lower arithmetic intensity (operations per memory access) [48, 62] and thus is memory-bound.*

2.3 GPU Architecture

We abstract the GPU architecture as shown in Figure 2(b), which logically consists of multiple blocks, each containing a tensor core, and a fast but limited on-chip memory (shared memory), while the large but slow global memory is shared among all blocks [1]. Therefore, two levels of parallelism can be exploited: 1) **intra-block parallelism** within each block, which is typically achieved through tensor cores and can be further accelerated by shared memory, and 2) **inter-block parallelism** across multiple blocks, which is needed to balance the workload among blocks.

2.4 FlashAttention and FlashDecoding

FlashAttention: FlashAttention [7] and its successor FlashDecoding [8] are highly optimized CUDA kernels for attention computation, designed to leverage the GPU's memory hierarchy and parallelism capabilities. As shown in Figure 2(c) and Algorithm 1 (Line 3-6), FlashAttention decomposes and parallelizes the attention operation in the *1. batch, 2. head, 3. query sequence and 4. KV sequence dimensions* into multiple blocks, where each block can fit into the shared memory. Subsequently, in lines 7-11, the partial attention computation of each block is performed in shared memory, which takes $Q \in \mathbb{R}^{\frac{n_q}{b_q} \times d}$ and $K, V \in \mathbb{R}^{\frac{n}{b_k} \times d}$ to generate the partial output O. Finally, in line 12, we reduce the partial outputs in the KV sequence dimension to obtain the final output O.

Algorithm 1 FlashAttention

Input: Q, K, V
Output: O
for $seq = 1$ **to** bs **in parallel do**
  for $head = 1$ **to** h **in parallel do**
    for $i = 1$ **to** b_q **in parallel do**
      for $j = 1$ **to** b_k **in parallel do**
        // Executed in shared memory
        $Q = \mathbf{Q}[seq, (i-1)\frac{n_q}{b_q} : i\frac{n_q}{b_q}, head, :]$
        $K = \mathbf{K}[seq, (j-1)\frac{n}{b_k} : j\frac{n}{b_k}, head, :]$
        $V = \mathbf{V}[seq, (j-1)\frac{n}{b_k} : j\frac{n}{b_k}, head, :]$
        $O[j] = \text{softmax}(\frac{QK^T}{\sqrt{d}})V$
      $\mathbf{O}[seq, (i-1)\frac{n_q}{b_q} : i\frac{n_q}{b_q}, head, :] = \text{reduce}(O)$
return O

It is worth noting that FlashAttention and FlashDecoding are designed for regular 4D tensors, which makes parallelism and partitioning easier.

2.5 Prefix-sharing

Many real-world workloads exhibit opportunities for prefix sharing. Several notable examples include:

- **Document Question Answering (QA) [12, 23, 51].** Users may ask multiple questions about the same document. For instance, in the LooGLE [23] dataset, the average prompt length is 23474 tokens, and the sharing rate is 91%.
- **Tool-use [18].** If multiple requests share the same tool usage, they can share the prefix of the tool description and the tool usage instructions. The ToolBench dataset [18] has an average prompt length of 1835 tokens and a sharing rate of 85%.
- **Few-shot Prompting [6]**. This technique often involves prepending identical instructions or examples (e.g., demonstrations of tool usage) to various distinct prompts.
- **Self-consistency [46]**. It uses a standard Chain-of-Thought (CoT) few-shot prompt, and samples a diverse set of reasoning paths. This initial CoT prompt acts as the shared prefix for multiple sampling iterations.
- **Tree-of-thoughts [55].** It explores multiple solution paths by building a tree of intermediate steps, where each branch represents a possible reasoning path. Its tree structure allows prefix sharing, as sibling nodes reuse common parent computations.
- **Speculative Decoding [29].** Within the verification phase of speculative decoding, the generation process can form tree-structured queries where nodes representing sequential tokens share common ancestor paths, enabling prefix sharing.

Existing research [20, 34, 47, 58, 60, 61] leverages KV cache reuse for requests sharing the same prompt prefixes, thereby accelerating the prefill phase and reducing memory consumption. They typically maintain the KV cache in a tree fashion, where each node corresponds to a prefix. However, when accessing the KV cache, the system still assumes *a logical 4D tensor structure for the Key and Value tensors, thus still suffering from duplicated global memory accesses.*

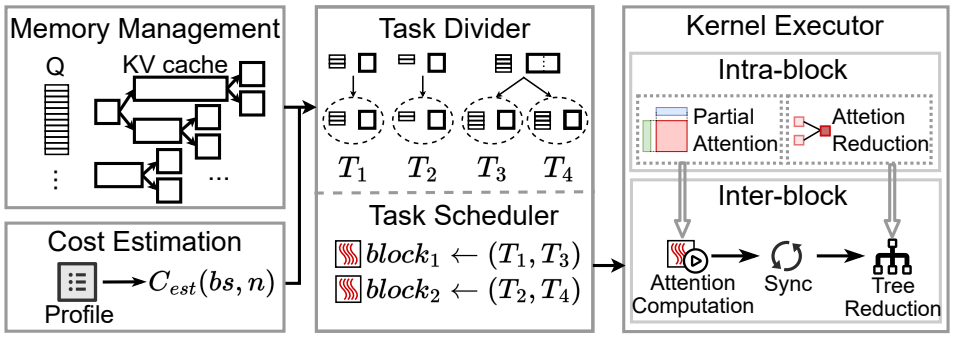


Fig. 3. Overview of CoDec.

These observations surface two key challenges in prefix-shared decoding—(i) kernel design over an irregular KV-cache forest and (ii) workload imbalance under highly skewed per-node workloads—which we address in Section 4 and Section 5, respectively.

3 Overview

3.1 Requirement

To achieve efficient prefix-shared decoding, the developed kernel should satisfy the following requirements:

IO Efficient: As the decode stage is memory-intensive, the prefix-shared decoding kernel should be able to minimize the global memory access overhead by leveraging the shared memory to conduct the attention computation between the KV cache of shared prefix tokens and the query tensor of these requests. In addition, the developed kernel should have sufficient parallelism for inter-block parallelism and only introduce limited extra overhead for synchronization and reduction.

Workload Balance: As the computation is shifted from regular 4D tensor to tree of 3D tensors, the workload distribution among different blocks is unbalanced. The developed kernel should be able to balance the workload among different blocks, to put it in another way, the divided subtasks should have a similar workload.

3.2 System Architecture

Memory Manager (Section 4.1): The KV cache of the current running batch is materialized as a tree of tensors in the global memory, where each node maintains the KV cache of a chunk of tokens shared by multiple requests or owned by a single request. The queries of the requests are materialized as a query tensor. Moreover, we maintain a lightweight index that links each query to the relevant nodes in the KV-cache tree, enabling the kernel to directly locate and coalesce reads to shared-prefix KV blocks.

Kernel Executor (Section 4.2, 4.3): The kernel executor exposes both intra-block and inter-block parallelism. At the intra-block level, we reuse and adapt two primitives—partial attention computation (PAC) and partial output reduction (POR)—to efficiently compute and aggregate partial results. At the inter-block level, CoDec schedules PAC across the prefix-tree blocks and then performs a parallel, tree-structured reduction based on POR to merge partial outputs per query, avoiding the overhead of launching many small, sequential reduction kernels.

Cost Estimator (Section 5.2): The cost estimator predicts the execution time of PAC for a given (query, KV-block) shape. Because PAC may be either compute-bound or memory-bound depending

on tensor shapes and GPU characteristics, CoDec uses lightweight micro-benchmark profiling on the target GPU/model and interpolates to estimate costs for unprofiled workloads.

Task Divider and Scheduler (Section 5.1): Given a decoding batch, the workload across prefix-tree nodes is highly skewed. CoDec uses the cost estimator to divide and assign work with a global view of the prefix tree, aiming to balance block-level execution time while avoiding overly fine-grained partitioning that would increase scheduling and reduction overhead.

4 GPU Kernel Design

In FlashAttention/FlashDecoding, KV tensors follow a regular 4D layout that naturally supports efficient materialization, block partitioning, and reduction. In prefix-shared decoding, the KV cache becomes a forest of per-prefix 3D tensors, which complicates KV/query indexing, increases task-division complexity, and requires tree-structured reduction to merge partial results.

We first introduce our compute-centric KV cache management for facilitating the attention computation in the prefix-shared decoding kernel. Then, we formulate two essential intra-block kernel primitives: the partial attention computation kernel and the partial output reduction kernel, which serve as the building blocks of our prefix-shared decoding kernel. Finally, we introduce the inter-block kernel executor, which orchestrates the parallel execution of these intra-block kernel primitives to maximize computational efficiency.

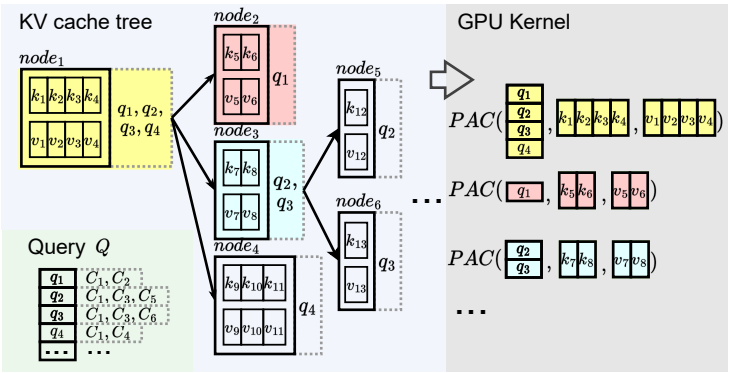


Fig. 4. KV cache forest.

4.1 KV Cache Management

Different from the traditional KV cache management systems, e.g., PagedAttention [22] targeting for maintaining the KV cache in the GPU memory, our prefix-shared decoding kernel has further responsibilities to support the prefix-shared decoding, which requires efficient indexing for the further partial attention computation and reduction operations.

Formal model and notation. Let $\mathcal{R} = \{r_1, \ldots, r_B\}$ be the batch of requests. Each request r has a current query vector $q_r \in \mathbb{R}^{d_k}$, and a (possibly shared) prefix token sequence $P(r) = (t_1, \ldots, t_{\ell(r)})$. We maintain a forest (with a virtual root) $\mathcal{F} = (\mathcal{N}, \mathcal{E})$ of KV-cache nodes, where each $n \in \mathcal{N}$ stores tensors

$$K_n \in \mathbb{R}^{|n| \times d_k}, \qquad V_n \in \mathbb{R}^{|n| \times d_v},$$

and an ordered token index set $\mathrm{Tok}(n) = \{1, \ldots, |n|\}$. An edge $(p \longrightarrow c) \in \mathcal{E}$ encodes p is a prefix of c (write $p \prec c$). A virtual root $\varnothing$ connects all prefix roots.

Define the *prefix path* of r:

$$\pi(r) = (n_1, \ldots, n_{m(r)})$$
$$\text{with} \quad \varnothing = n_0 \prec n_1 \prec \cdots \prec n_{m(r)}$$
$$\text{s.t.} \quad \text{concat}\big(\{K_{n_i}\}\big) \text{ matches } P(r)$$

All queries are stacked into $Q \in \mathbb{R}^{B \times d_k}$ with $Q[r,:] = q_r^\top$. Let the global token address space be

$$\Omega = \{(n, i) \mid n \in \mathcal{N},\ i \in \text{Tok}(n)\},$$

and a bijection (logical flattening) $\kappa : \Omega \xrightarrow{\sim} \{1, \ldots, L_{\text{tot}}\}$ that orders tokens across nodes; this induces logical tensors K, V by

$$K[\kappa(n,i),:] = K_n[i,:], \qquad V[\kappa(n,i),:] = V_n[i,:].$$

(We never physically materialize K, V monolithically; κ is index-level.)

Tree-based KV cache management. As shown in Figure 4, we manage the KV cache as a tree of tensors, where each node in the tree represents a chunk of KV cache, and the edge between the father and child nodes represents the relationship between the two chunks, where the father node is the prefix of the child node. Moreover, the queries from all requests are consolidated into a single query tensor, with each row corresponding to an individual request's query. Notice, Figure 4 illustrates the example of all requests sharing the same prefix in node 1, while in practice, two or more prefixes may be shared by different requests. Therefore, we introduce a virtual root node to represent the root of the tree, which connects all the prefixes of the requests. This virtual root node allows batching different prefixed requests together, which allows the kernel to support even the non-prefix-shared decoding.

In addition to the tensor enclosed in solid lines, we also maintain the following index structures enclosed in dashed lines, which record the bijective mapping between the KV cache and the query tensor.

Tensors preparation for partial attention computation. As shown in the right part of Figure 4, the queries shared the same prefix maintained in the KV cache node actually form a partial attention computation. Therefore, for each KV cache node, we need to maintain the *set of queries* of the requests that share the same prefix with this node, which facilitates the aggregation of the queries that share the same prefix, allowing further partial attention computation between tensors formed by the queries and the corresponding KV cache node. Instead of preparing the query tensor in the global memory, the query tensor can be aggregated in thread block shared memory during partial attention kernel launch, reducing memory overhead.

Formal per-node assembly. For node n, define the per-node query tensor

$$Q^{(n)} \stackrel{\text{def}}{=} Q[I_n,:] \in \mathbb{R}^{|\mathcal{R}(n)| \times d_k}.$$

We assemble $Q^{(n)}$ on-chip (shared memory) at kernel launch:

$$Q^{(n)}[a,:] \leftarrow Q[i_a,:] \quad (a = 1, \ldots, |I_n|),$$

and stream tiles of (K_n, V_n) to shared memory for the GEMM and value-weighted reduction.

Indexing of partial results for each query. As introduced in Section 2.2, the softmax operation inherently works globally across each row of the attention score tensor, essentially operating at the query level. This characteristic introduces a specific indexing requirement: for each query, the system needs to maintain a record of which KV cache nodes constitute its prefix. This query-specific tracking enables correct indexing and retrieval of partial attention computation results, forming a critical component when aggregating across multiple partial computations.

Streaming softmax across nodes. Our goal is to compute attention for each query $r \in \mathcal{R}$ without materializing the global score vector: we stream over nodes (blocks) and accumulate the softmax numerator and denominator on the fly. Let J_r be the set of nodes visible to r. When visiting a node $n \in J_r$, we compute the scaled scores

$$s_{r,n} = \frac{1}{\sqrt{d_k}} Q[r,:] K_n^\top \in \mathbb{R}^{|n|},$$

and apply the visibility mask to obtain

$$\tilde{s}_{r,n}[j] = \begin{cases} s_{r,n}[j], & \text{if } (n,j) \text{ is visible to } r, \\ -\infty, & \text{otherwise.} \end{cases}$$

We then perform *local* stabilization within the node: with the nodewise maximum $m_{r,n} = \max_j \tilde{s}_{r,n}[j]$, define

$$d_{r,n} = \sum_j e^{\tilde{s}_{r,n}[j] - m_{r,n}}, \qquad u_{r,n} = \sum_j e^{\tilde{s}_{r,n}[j] - m_{r,n}} V_n[j,:] \in \mathbb{R}^{d_v}.$$

Here $d_{r,n}$ is the node's contribution to the softmax denominator, and $u_{r,n}$ is the vector contribution to the numerator. Masked (invisible) positions are set to $-\infty$, hence contribute 0 after exponentiation.

To avoid overflow/underflow when merging across nodes, we maintain *per-query accumulators*

$$M[r] \leftarrow -\infty, \quad D[r] \leftarrow 0, \quad U[r,:] \leftarrow \mathbf{0} \in \mathbb{R}^{d_v}.$$

When incorporating node n, we use a common log-sum-exp reference $\hat{M} = \max\left(M[r],\, m_{r,n}\right)$ and combine the old and new contributions in the same exponential frame.

$$M[r] \leftarrow \hat{M}.$$

Intuitively, $M[r]$ tracks the running global maximum logit, while $D[r]$ and $U[r,:]$ are the denominator and numerator evaluated relative to that maximum. Iterating over $n \in J_r$ in this way preserves numerical stability and requires memory only proportional to the current node.

4.2 Intra-Block Kernel Primitive

Following the KV cache management, we abstract two necessary operations, computing the partial attention within each KV cache node and reducing the partial attention computation results between the KV cache nodes, into two intra-block kernel primitives, namely, the partial attention computation (PAC) kernel and the partial output reduction (POR) kernel. As the GPU architecture suggests (as shown in Figure 1(b)), the intra-block kernel primitives are executed upon the thread blocks configured with on-chip shared memory, our intra-block kernel primitives are designed to be executed in shared memory, which can significantly reduce the global memory access overhead.

Algorithm 2 PAC

1: **Input:** Q, K, V
2: **Output:** O
3: $S \leftarrow Q \cdot K^T / \sqrt{d}$
4: $S \leftarrow \text{SOFTMAX}(S)$
5: $O \leftarrow S \cdot V$
6: **return** O

Algorithm 3 POR

1: **Input:** O_1, O_2, m_1, m_2, s_1, s_2
2: **Output:** O
3: $m \leftarrow \max(m_1, m_2)$
4: $s \leftarrow s_1 e^{m_1 - m} + s_2 \cdot e^{m_2 - m}$
5: $O \leftarrow \frac{O_1 \cdot s_1 \cdot e^{m_1 - m} + O_2 \cdot s_2 \cdot e^{m_2 - m}}{s}$
6: **return** O

Partial attention computation (PAC) kernel: As the name suggests, the partial attention computation kernel is responsible for performing attention computation between a query sub-tensor ($Q \in \mathbb{R}^{n_q \times d}$) and its corresponding KV cache sub-tensor ($K, V \in \mathbb{R}^{n \times d}$), where n_q queries in the query sub-tensor share the same prefix with length n maintained in the KV cache sub-tensor.

We observe that the computation of the PAC kernel is exactly the same as the computation of the attention operation, except that the input query tensor in the PAC kernel is sourced from multiple requests while the input query tensor in the ordinary attention operation are sourced from different tokens of the same request. We efficiently support the intra-block partial attention computation as demonstrated in Algorithm 2, requiring only minimal modifications to the FlashAttention kernel. Instead of limiting the memory consumption of query, key, and value tensors to shared memory, our intra-block kernel primitives further partition the partial attention computation to leverage the shared memory, and sequentially process partitioned computations, thereby supporting larger workloads while maintaining the memory efficiency by leveraging the shared memory.

To further reduce the memory access overhead, we also optimize the data loading pattern of the PAC kernel to better leverage the shared memory. Specifically, instead of loading the key and value tensors from the global memory for each partial attention computation, we load the key and value tensors into the shared memory once, and then reuse them for multiple query. This optimization can significantly reduce the memory access overhead, especially when the ratio of GQA is high. Please note that this optimization also reduces the inefficiency inside Tensor Cores, as the padding of the query tensor is reduced.

Partial output reduction (POR) kernel: As the operation between a query and its corresponding KV cache will be partitioned into several partial attention computations as the KV cache is divided into several KV cache nodes, we need to merge the results of these partial attention computations to obtain the final output of the query. Instead of reducing the whole results, the POR kernel is a binary reduction operation, which merges two partial attention computation results $O_1 \in \mathbb{R}^{n_q \times d}$ and $O_2 \in \mathbb{R}^{n_q \times d}$ sourced from different KV cache nodes of the same query set.

As shown in Algorithm 3, the POR kernel takes O_1, O_2, and their corresponding max attention scores m_1, m_2 and the sum of exp attention scores s_1, s_2 as input, and outputs the final output O of the query. Line 3 computes the maximum attention score m of the two partial attention computation results while line 4 computes the sum of the attention scores s of the two partial attention computation results, both of which are used to normalize the final output O. Subsequently, line 5 renormalizes the two partial attention computation results O_1 and O_2 and merges them into the final output O. As the size of the output O can be easily fitted into the shared memory, POR kernel by default will be executed in the shared memory.

4.3 Inter-Block Launching and Tree-Reduction

On top of the intra-block kernel primitives, we develop the inter-block kernel executor, which is responsible for executing the intra-block kernel primitives in a parallel manner to conduct the prefix-shared attention computation for a batch of requests.

Algorithm 4 illustrates how to sequentially launch the intra-block kernel primitives to perform the attention computation. It mainly consists of two steps: 1) launching the PAC kernel for each KV cache node (lines 4-6), 2) conducting the tree reduction to merge the results for each query (lines 7-8).

The PAC kernel launching is quite straightforward, as each KV cache node and its corresponding query tensor can be easily indexed by the KV cache management system. Given the computations of the PAC kernel are independent, we can leverage embarrassingly parallelism in line 4 to launch the PAC kernel for each KV cache node in parallel. Moreover, a synchronization is conducted

Algorithm 4 CoDec

1: **Input:** $\mathbf{Q}$, $(\mathbf{K}, \mathbf{V})$
2: **Output:** $\mathbf{O}$
3: Initialize $\mathbf{O}_{tree}$ with the tree structure of $(\mathbf{K}, \mathbf{V})$
4: **for** $(K, V) \in (\mathbf{K}, \mathbf{V})$ **do**
5: Aggregate the tensor Q from query index in (K, V)
6: $\mathbf{O}_{tree}[(K, V).index] \leftarrow PAC(Q, K, V)$
7: **for** $O \in \mathbf{O}_{tree}$ **do**
8: $\mathbf{O}[O.query\_index] \leftarrow POR(\mathbf{O}[O.query\_index], O)$
9: **return** $\mathbf{O}$

to ensure that all partial attention computation results are prepared before conducting the tree reduction operation.

As shown in line 6, the partial attention computation results of the PAC kernel are stored in a tree structure same as the KV cache management system. Therefore, the reduction operation should be conducted in a tree structure, which introduces the challenge in parallelization.

Parallelization of the tree reduction operation. We notice that the reduction operation satisfies the *associative and commutative properties*. Specifically, recalling Algorithm 3, the reduction operation between two partial attention computation results O_1 and O_2 is independent of the order of the reduction operation, i.e., associative ($POR(O_1, O_2) = POR(O_2, O_1)$) and commutative ($POR(POR(O_1, O_2), O_3) = POR(O_1, POR(O_2, O_3))$). These properties allow us to reorganize the reduction operation order, which facilitates the parallelization. Moreover, lines 7-8 in Algorithm 4 implicitly indicate that the *reduction operation of different queries is independent*, no matter how the tree is structured. These two observations suggest that we can transform the tree reduction operation into bs independent series of reduction operations, where bs is the number of queries in the batch. Moreover, the reduction operation of non-adjacent edges in the series of each query can be conducted in parallel, as the commutative property of the reduction operation allows us to change the order of the reduction operation. Hence, we can easily speed up the tree reduction by exploiting parallelism in two dimensions: 1) parallelizing the reduction operation of different queries, 2) parallelizing the reduction operation of different nodes in the tree by replicating $\mathbf{O}$ and conducting an addition reduction on the replicated $\mathbf{O}$.

Complexity analysis. The IO complexity of CoDec can be denoted as $O(h \cdot d \sum_{i=1}^{node\_num} n[i])$ while the IO complexity of FlashAttention is $O(h \cdot d \sum_{i=1}^{node\_num} n[i] \times n_q[i])$, where we ignore the cost of loading the query tensor and writing the output tensor, as it is negligible compared to the cost of loading the KV cache tensor (i.e., $n_q[i] \ll n[i]$). Intuitively, given $\overline{n_q}$ as the weighted average of the shared ratio of the KV cache, i.e., $\overline{n_q} = \frac{\sum_{i=1}^{node\_num} n[i] \times n_q[i]}{\sum_{i=1}^{node\_num} n[i]}$ the IO complexity of CoDec is about $\overline{n_q}$ times lower than that of FlashAttention. Regarding the computation complexity, CoDec is the same as FlashAttention.

5 Workload Balance

For a KV-cache node, both the KV length (prefix length) and the number of associated queries (degree of sharing) can vary widely across nodes. As a result, naively launching one PAC kernel per node leads to severe inter-block load imbalance and GPU under-utilization; however, overly fine-grained splitting increases scheduling and reduction overhead and may underutilize tensor cores within each block.

In this section, we first formulate the optimization problem of task division and scheduling, which is np-hard, and propose a heuristic solution through pruning. Recognizing the inaccurate of theoretical cost estimation, we further propose a profile-based estimator.

5.1 Task Division and Scheduling

An intuitive approach is to further divide the KV cache node into several sub-nodes as well as the queries into several sub-queries, and then assign each sub-node to a thread block. However, determining the granularity of this division presents significant challenges. On the one hand, fine-grained task division can achieve better workload balance, but it may result in a large number of tasks, which will introduce additional scheduling and reduction overhead, and a fine-grained task can also lead to resource under-utilization within each thread block due to insufficient workload for tensor core in each block. On the other hand, coarse-grained task divisions may still suffer from workload imbalance, which also leads to under-utilization of GPU resources due to inter-block stallings.

Therefore, we formulate the task division and scheduling problem as an optimization problem, where we aim to find the optimal task division and scheduling strategy to minimize the execution time of the slowest thread block.

Division and scheduling formulation. The partial attention computations can be modeled as a set of tasks, where each task is a tuple $(Q[i], K[i], V[i])$. As the dimension of feature d is fixed, we can ignore it in the task division formulation, thereby we use $\mathbf{T}[i] = (n_q[i], n[i])$ to represent i-th task, where $n_q[i]$ is the number of queries in the i-th task and $n[i]$ is the sequence length of the KV cache node in the i-th task. We use t to denote the number of KV cache nodes, i.e., the number of tasks. For each task $\mathbf{T}[i]$, we can divide it both horizontally (i.e., in the query dimension, whose number of horizontal slices denotes $b_q[i]$) and vertically (i.e., in the KV cache dimension, whose number of vertical slices denotes $b_k[i]$). Therefore, we aim to find the optimal task division strategy $\{(b_q[i], b_k[i])\}_{i=1}^{t}$ to minimize the total execution time of all tasks.

In addition to task division, we also need to consider task scheduling or task assignment strategy. Assuming we have m thread blocks, we need to assign the tasks to the thread blocks. The task assignment strategy can be represented as a tensor $\mathbf{A} \in \mathbb{N}^{m \times t}$, where $\mathbf{A}[i, j]$ is the number of divided sub-tasks of task j assigned to thread block i, m is the number of thread blocks. To facilitate our discussion, we use $\mathbf{C}[j]$ to represent the estimated execution time of a sub-task of task j, which can be computed by the cost estimator will be introduced in Section 5.2.

Formally, we can formulate the task division and scheduling problem as follows:

$$\begin{aligned} \arg\min_{b_q, b_n, \mathbf{A}} \quad & Cost = \max_{i=1}^{m}\Big(\sum_{j=1}^{t} \mathbf{C}[j] * \mathbf{A}[i, j]\Big), \\ s.t. \quad & \forall j \in [1, t], \sum_{i=1}^{m} \mathbf{A}[i, j] = b_q[j] \cdot b_k[j]. \end{aligned} \tag{3}$$

The objective function is to minimize the maximum execution time of all thread blocks, where the cost of each thread block is the sum of the execution time of all subtasks assigned to this thread block. The constraint is to ensure that all subtasks of each task are assigned to the thread blocks.

Solver. The above problem is an advanced parallel task scheduling problem [16], which is NP-hard. Subsequently, we first simplify the problem, and then narrow down the search space by obtaining the lower and upper bounds of the *cost*, and finally exhaustively search the task division and scheduling strategy.

We observe that the $n_q \ll n$ in most cases as the KV cache is usually much larger than the number of queries. If we divide the task into the query dimension, the cost increases significantly, as it actually misses the opportunity to combine the KV cache memory access. Therefore, we set the number of horizontal slices $b_q[i]$ to 1, and focus on the vertical slices $b_k[i]$.

To further narrow down the search space, we easily find the following two properties inequalities. 1) Noticing that the sum cost of the sub-tasks is no less than the cost of the original task, and the max cost of each block is no less than the average cost of all blocks, we can reach the following inequality:

$$Cost \geq \frac{1}{m}\sum_{i=1}^{m}(\sum_{j=1}^{t}\mathbf{C}[j] * \mathbf{A}[i,j]) \tag{4}$$

Moreover, with more fine-grained task division, the average cost of all blocks will be larger, as the workload is not reduced, but the scheduling overhead is increased. This monotonicity and the inequality in Equation 4 can be used to determine the lower bound of the cost (denoted as $cost_l$) through binary search.

Therefore, we can narrow down the search space by setting the upper bound of the division number of each KV cache node as

$$b_k[i] \leq \lceil \frac{C_{est}(n_q[i], n[i])}{cost_l} \rceil, \tag{5}$$

where $C_{est}(n_q[i], n[i])$ is the estimated execution time defined in Section 5.2. This inequality restricts the further division of the KV cache node when the cost is lower than the average cost, as under such conditions, subtasks are enough to saturate the GPU's block-level parallelism, while further division will lead to more overhead.

In practice, the equation 5 sets the division number of most tasks to 1, whose workload is significantly smaller than the average cost. For example, in documented question-answering tasks, despite the shared document KV cache node ($n \approx 10k$), the workload of the question KV cache node for each request ($n \approx 50$) is usually much smaller. Therefore, we gird search the division number of each KV cache node and choose the optimal division.

5.2 Cost Estimation

We observe that *the execution cost of partial attention computation varies from the theoretical result.* It is easy to compute the theoretical workload of the partial attention computation given $Q \in \mathbb{R}^{n_q \times d}$ and $K, V \in \mathbb{R}^{n \times d}$. The computation mainly involves two matrix multiplications, i.e., QK^T and AV, where A is the attention score tensor. The theoretical workload of the first matrix multiplication is $O(n_q \times n \times d)$, and the second matrix multiplication is $O(n_q \times n \times d)$, resulting in a total theoretical workload of $O\left(n_q \times n \times d\right)$. Moreover, the global memory access is the sum of the memory access of Q, K and V, which is $O\left((n_q + 2n) \times d\right)$. The divergence between the computation and memory access makes the execution cost hard to estimate. Moreover, the kernel has a constant launch overhead, which is independent of the workload. Therefore, for the small workload, the execution cost is dominated by the kernel launch overhead, while for the large workload, the execution cost is dominated by the computation and memory access.

As revealed in Table 2, different shapes of the tensors will lead to different execution costs due to the varying hardware resource utilization. When the workload is small, the execution time is dominated by the kernel launch overhead. For small n_q and large n, the PAC kernel is memory bound, thus cost scales almost linearly with n, while for large n_q and n, the PAC kernel is compute bound.

Table 2. Thread block execution time (ms), $d = 128$.

n \ n_q	1	2	5	10	20	50	100
512	0.036	0.035	0.036	0.043	0.048	0.074	0.112
1,024	0.043	0.043	0.044	0.054	0.062	0.109	0.122
2,048	0.060	0.059	0.059	0.079	0.094	0.124	0.145
4,096	0.092	0.092	0.093	0.126	0.147	0.156	0.183
8,192	0.156	0.157	0.156	0.199	0.189	0.195	0.266
16,384	0.283	0.282	0.283	0.301	0.303	0.471	0.746

Therefore, we propose a profile-based approach to estimate the execution cost of the partial attention computation between each KV cache node and its corresponding queries. With a given hardware configuration and a given model (i.e., the dimension of the feature d), we observe that only n and n_q are the two parameters that affect the execution time of the PAC kernel. Therefore, before deploying the model, we can profile the PAC kernel with various sizes of the KV cache node n and various numbers of queries n_q, and record the execution time. For the unprofiled computation, we use interpolation to estimate the execution time. After profiling, we have the following cost estimation function:

$$C_{est}(n_q, n), \tag{6}$$

which is the estimated execution time of the partial attention computation between a KV cache node with sequence length n and n_q queries.

Recall the cost in Equation 3, for the cost of a subtask of task $T[i]$, we can estimate the execution time as:

$$\mathbf{C}[j] = C_{est}(\frac{bs_j}{v_j}, \frac{n_j}{h_j}). \tag{7}$$

6 Implementation

We implement the CoDec kernel module in CUDA/C++ on top of NVIDIA CUTLASS (about 1,700 LOC) and the task-division module in C++ (about 300 LOC). The implementation is modular to facilitate extension to new models and attention variants. The kernel supports widely used attention layouts in modern LLMs, including MHA, MQA, and GQA.

CoDec Kernel Module. The CoDec kernel takes the query tensor, paged KV cache, and decoding metadata (e.g., block tables and context lengths) as inputs and writes attention outputs to the designated output tensor. It follows the same paged KV-cache layout as PagedAttention [22] and exposes an attention interface compatible with FlashDecoding, making integration into vLLM straightforward (i.e., swapping the attention backend while reusing the existing batching and memory manager). The kernel is implemented with CUTLASS to leverage fine-grained pipelining across memory and compute.

For models using multi-head latent attention (MLA), CoDec can be extended by first reconstructing per-head KV blocks from the latent representation and then applying the same prefix-aware attention and reduction pipeline.

CoDec Task Division Module. To maximize the efficiency of the task division module, we implement it as a C++ module that interfaces with the main decoding loop. This module monitors the decoding process and dynamically divides tasks based on the current workload and resource availability. To reduce overhead, we perform task division every few decoding steps rather than at every step. Empirically, the partitioning overhead accounts for 1.3%–2.5% of the total attention

time in our evaluation, and the parallel reduction contributes less than 10% overhead relative to partial attention computation under typical shared-prefix workloads.

7 Evaluation

In this section, we evaluate the performance of CoDec on various tasks. In summary, we want to answer the following questions:

- **Section 7.2: What is the benefit of using CoDec?** We compare CoDec with the SOTA attention kernel, FlashAttention [7] and FlashDecoding [8], in terms of attention operation time, end-to-end time, and global memory IO.
- **Section 7.3: How does each optimization contribute to the performance?** We conduct an ablation study to analyze the contribution of each optimization in CoDec.
- **Section 7.4: How do the specific design choices impact CoDec's performance?** We analyze the impact of key design decisions in CoDec, particularly focusing on the optimal division granularity for different sequence lengths.
- **Section 7.5: What is the overhead of task division?** We quantify the CPU overhead of computing a division plan and show how it scales with batch size.
- **Section 7.6: How does CoDec perform on different GPUs?** We evaluate the performance of CoDec on different GPUs, including NVIDIA H800, A100, RTX 4090, A30, and RTX A6000.

7.1 Experimental Setup

CoDec is implemented in around 2,000 lines of CUDA/C++. By default, we use Qwen3-4B, which has 32 query heads, 8 key/value heads, and head dimension 128. Unless otherwise specified, we run experiments on a single NVIDIA A100 GPU (40GB, PCIe) with CUDA Toolkit 11.8 (runtime version 12.2), vLLM 0.6.6, and Python 3.10. We report the mean over 3 runs. To study generality, we additionally evaluate across attention layouts (MHA/MQA/GQA) and multiple model families/sizes (Section 7.7).

Workloads. We evaluate both controlled synthetic prefix trees and a real-world long-context dataset. For synthetic workloads, we use controlled prefix-sharing trees; the detailed workload specifications are described in Section 7.2. For real-world evaluation, we use the LooGLE dataset and materialize shared prefixes by grouping queries that share the same document context.

Baselines and metrics. We compare against FlashDecoding [8] (decode-stage attention kernel) and FlashAttention [7] where applicable. We also include FlashInfer's multilevel cascade attention [57] in a complementary shared-prefix throughput comparison (Figure 8). We report attention-kernel time (CUDA events), end-to-end decoding time per output token (TPOT) in comparison with vLLM, and global memory traffic of attention kernels measured via GPU profiling counters. Finally, we discuss the overhead of CoDec's workload partitioning and tree reduction: while the benefit naturally diminishes when prefix sharing is limited, the partitioning overhead remains small in practice because we amortize partitioning by reusing a division plan for multiple decoding steps (Section 6).

7.2 Comparison with SOTA

To further reveal the performance characteristics of CoDec, we conduct a series of experiments to evaluate the impact of different workloads on the performance of CoDec. We consider the following workloads:

Workload. By default, we consider the 2-level tree structure, where the root node is the prefix shared by all requests, and the leaf nodes are the KV cache of each request. This is a common case

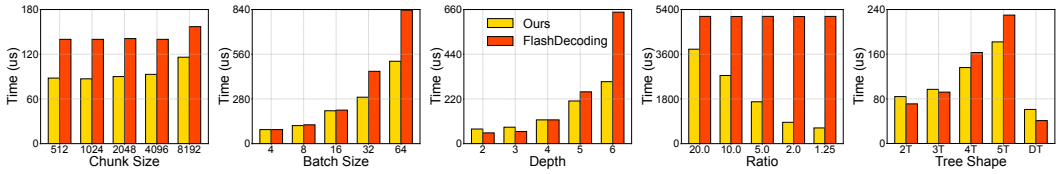


Fig. 5. CoDec vs. FlashDecoding on execution time.

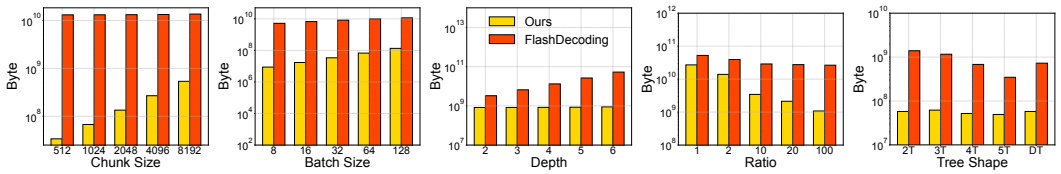


Fig. 6. CoDec vs. FlashDecoding on global memory access.

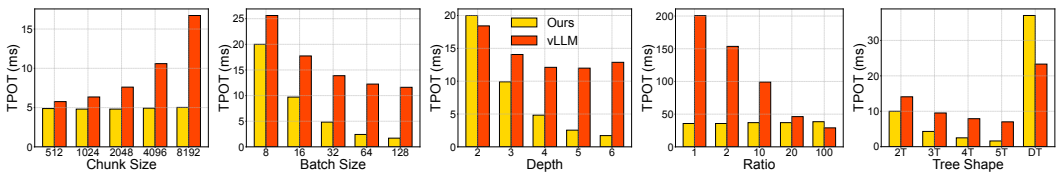


Fig. 7. CoDec vs. vLLM on end-to-end time.

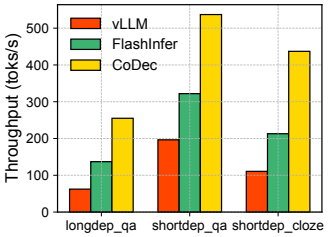
in document QA tasks, where all requests share the same document. Generally, we consider the following workloads:

- **Varying sequence length:** Fixing the full binary shared prefix tree with depth 2. We vary the sequence length of the non-shared prefix from 512 to 8,192 tokens.
- **Varying batch size:** Fixing the full binary shared prefix tree with depth 2 and root node context length of 120k, we vary the batch size, i.e., the number of requests.
- **Varying tree depth:** In this workload, we choose the full binary tree structure, where each node has two children, and vary the tree depth from 2 to 6.
- **Varying shared prefix ratio:** We vary the shared prefix ratio by controlling the number of shared tokens in the default 2-level tree structure with a total context length of 120k. The shared prefix ratio is defined as the number of shared tokens divided by the total number of tokens in the KV cache tree.
- **Varying tree shape:** We consider 1) binary tree (2T), 2) ternary tree (3T), 3) quaternary tree (4T), 4) quinary tree (5T), each with the same workload. Moreover, we also consider 5) degenerate tree (DT), where only the left nodes have children.

Metrics and SOTA. We evaluate CoDec in terms of 1) attention-kernel execution time, 2) global memory access in the attention kernel, and 3) end-to-end latency, i.e., TPOT (Time Per Output Token) in decoding. Regarding attention-kernel execution time and global memory access, we compare CoDec with FlashDecoding [7] as provided by FlashAttention 2.7.4, which is a strong baseline for long-context decoding. For end-to-end latency, we compare against vLLM 0.6.6 [22] using a PyTorch 2.6.0 prototype that implements CoDec while following the same paged-KV metadata layout.

Dataset	Category	Average Tokens	Task
arXiv	Phys., Math,etc.	20,887	Summarization
Wiki	Hist., Polit.,etc.	21,017	short dep. QA long dep. QA
Scripts	Action, Drama,etc.	36,412	short dep. Cloze long dep. Cloze

(a) LooGLE Dataset



(b) Throughput

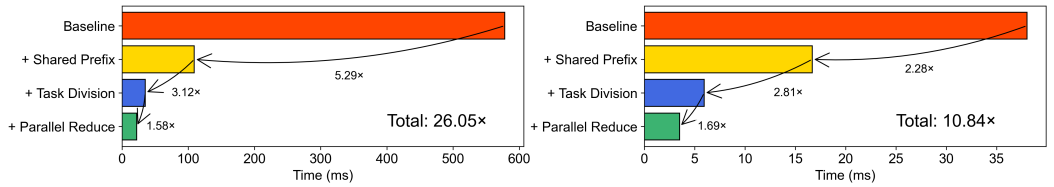
Fig. 8. CoDec vs. SOTA on real-world dataset.

Attention Execution Time. As shown in Figure 5, CoDec outperforms FlashDecoding up to 3.6× and averages 1.9× speedup across all workloads. We observe that a larger shared prefix results in a more significant speedup, and the case where the shared-to-unique ratio is 100 : 1 exhibits the highest speedup. Moreover, given the same shared prefix percentage, the speedup shows a trend of increasing with the decreasing workload size, which is because CoDec increases the workload in each subtask, resulting in better resource utilization. Interestingly, irregular workloads, such as 2.9×, exhibit a more pronounced speedup, compared to regular workloads, such as 1.77×. This is because CoDec can better balance the workload among different subtasks, leading to more efficient resource utilization.

Global Memory Access. Figure 6 shows the global memory access of CoDec and FlashDecoding, which verifies the performance gain of CoDec. The global memory access of CoDec is significantly lower than FlashDecoding across all workloads (14.66-409.80× lower), with an average reduction of 120.85×. Moreover, the same memory reduction does not always lead to the same performance gain, as shown in Figure 6 and Figure 5, which is attributed to the workload balance and scheduling strategy.

End-to-End Latency. We also evaluate the end-to-end latency of CoDec and vLLM in Figure 7. Our shared prefix contains both coding problems as well as code snippets as few-shot examples of question/answer pairs. Our benchmark uses the LooGLE dataset to let the model solve competitive programming problems on a single NVIDIA A100 PCIe-40G. We implement shared prefix tree in PyTorch, caching the shared prefix KV node in each attention layer as its KV cache component. The end-to-end latency of CoDec has an average latency reduction of 3.75× compared to vLLM. We notice that the sequence length has a significant impact on the end-to-end latency. This is due to that the larger the sequence length, the heavier the attention computation, while the FFN computation is insensitive to the sequence length but only related to the batch size.

FlashInfer proposes multilevel cascade attention for shared-prefix batch decoding [57]. To compare with this stronger multilevel baseline, we run a micro-benchmark that varies the shared-prefix ratio while keeping the overall context length fixed. Figure 8 shows that CoDec consistently achieves lower latency than FlashInfer's multilevel cascade attention across different shared ratios. This advantage mainly comes from two design choices: (1) CoDec performs workload partitioning with a global view of the entire prefix tree (rather than dividing each prefix node independently), which mitigates imbalance; and (2) CoDec uses a parallel reduction strategy to aggregate partial results efficiently, avoiding the overhead of launching many reduction kernels when the tree contains a large number of prefix nodes.



(a) Balanced tree workload (2T case). (b) Unbalanced tree workload (DT case).

Fig. 9. Ablation Study.

7.3 Ablation Studies

To understand how each component contributes to the overall performance gains, we conduct an ablation study that isolates three optimizations: (1) combining KV-cache accesses via the shared-prefix tree, (2) workload partitioning, and (3) parallel execution and reduction. We evaluate on two representative workloads: a balanced full binary shared-prefix tree and an unbalanced degenerate tree, both with a maximum context length of 200k tokens.

The results are shown in Figure 9(a) and Figure 9(b). For the unbalanced tree workload, the latency drops from 38.0 ms without optimization to 3.5 ms with all optimizations applied, achieving a 10.8× speedup. Using only the prefix tree or partitioning yields 16.7 ms and 5.9 ms, respectively. For the balanced tree workload, the latency is reduced from 578 ms to 22.2 ms with all optimizations, resulting in a 26.1× speedup. Applying only the prefix tree or partitioning leads to 109.2 ms and 34.9 ms, respectively.

These results show that each technique contributes to reducing latency, and combining all three yields the largest speedup. Notably, the impact of workload balancing and parallelism is more significant for the balanced tree, due to its higher intrinsic computational load.

7.4 Impact of Division Granularity

To further reveal the impact of task-division granularity, we compare CoDec against a naive strategy that evenly splits each task into a fixed number of subtasks. This naive strategy ignores workload skew in the KV-cache tree and query distribution, and thus may either create imbalance (too few splits) or introduce excessive overhead (too many splits).

Figure 10 compares the naive approach under different division counts with CoDec's adaptive division and scheduling. We plot CoDec as a horizontal dashed line since it automatically determines an effective division granularity based on the observed workload distribution. When the division count is 1, the naive approach degenerates to the baseline that executes without any task division.

The experimental results show that our approach outperforms the best fixed-division strategy of the naive approach by 1.02-1.04× across the tested workloads. Compared to the naive approach without division, our approach achieves 3.20-4.39× speedup and averages 3.80× across all workloads. This demonstrates the effectiveness of our division and scheduling strategy in balancing the workload among different subtasks and reducing the overhead of kernel launch and synchronization.

7.5 Overhead of Task Division

Task division is computed on CPU and could become a latency concern when the shared-prefix ratio is low and the attention kernel is relatively fast. We therefore measure the end-to-end time of generating a division plan (i.e., cost estimation, pruning, and greedy scheduling) as we increase batch size. Figure 11 shows that the overhead grows with batch size due to the increased number of tasks induced by larger prefix trees, but remains within tens of milliseconds even at batch size

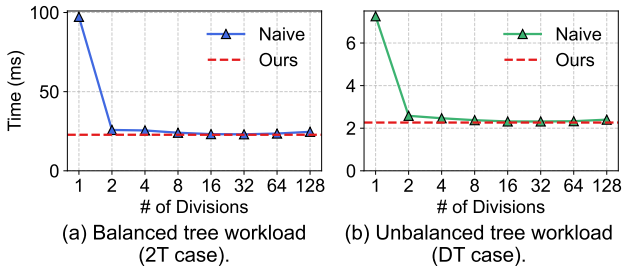


Fig. 10. Impact of division granularity.

64. In practice, we further amortize this cost by reusing a division plan for multiple decoding steps (Section 6), keeping the partitioning overhead small relative to attention execution time.

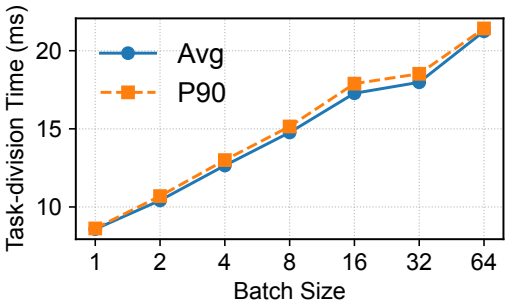


Fig. 11. CPU overhead of computing the task-division plan as batch size increases.

7.6 Performance in Varying GPU Cards

To evaluate cross-platform efficiency and performance consistency, we conduct experiments with a context length of 50K tokens on five modern GPUs.

As shown in Figure 12, CoDec consistently outperforms FlashDecoding [8] across all tested GPUs. On the H800, our method achieves a latency of 2.094 ms, compared to 9.900 ms for FlashDecoding, resulting in a 4.7× speedup. Even on lower-end GPUs such as the A6000, CoDec maintains a 15× advantage (2.869 ms vs. 43.048 ms).

The performance gap notably widens on GPUs with lower memory bandwidth. For example, FlashDecoding suffers on the A6000 (768 GB/s bandwidth), whereas our method degrades much more gracefully. This indicates that CoDec is less sensitive to hardware limitations, making it suitable for deployment across both data-center and consumer-grade GPUs.

7.7 Performance in Varying Models

Attention Variants. CoDec is designed to be compatible with widely used attention variants in modern LLMs, including MHA, MQA, and GQA. To evaluate this, we vary the grouping configuration of GQA (i.e., how many query heads share one KV head) and compare against FlashDecoding under the same shared-prefix workload.

As shown in Figure 13a, CoDec consistently reduces decoding latency across the tested configurations. The gain remains stable as the group size changes, indicating that our kernel design is not tied to a specific head layout and can generalize to different KV-sharing patterns encountered in real models.

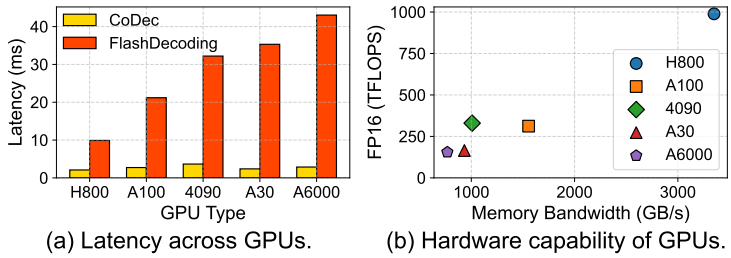


Fig. 12. Performance on diverse GPUs.

Model Sizes. We further test CoDec on multiple representative model families/sizes with different head configurations. Figure 13b shows that CoDec maintains consistent latency reduction across models, suggesting that the benefit mainly comes from eliminating redundant KV-cache reads and improving workload scheduling, rather than relying on a particular architecture.

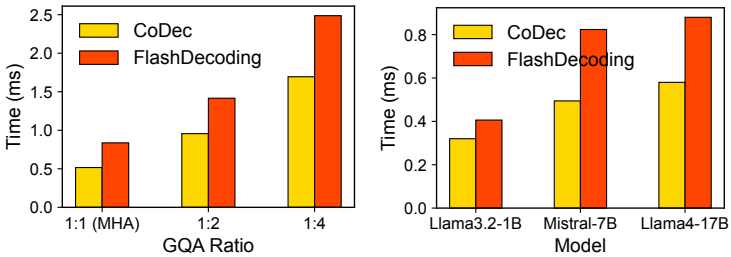


Fig. 13. Performance under different attention variants and models.

8 Related Work

Prefix caching. Prefix sharing is widely used to avoid redundant prefill computation by reusing KV states for common prompts. PagedAttention [22] provides a paged KV-cache abstraction that enables efficient memory management and sharing across requests. vLLM further introduces automatic prefix caching (APC) [45] to retain hot prefixes, while SGLang [60] uses a radix-tree-like structure to reuse historical KV states across structured program executions. Recent data-management work also studies KV-cache reuse policies and chunk-level caching for LLM serving. HotPrefix [25] schedules which prefix KV states to keep and when to promote them across GPU/CPU memory, while Cache-Craft [3] manages reusable chunk-caches for retrieval-augmented generation (RAG) workloads. These systems focus on improving KV reuse and cache management primarily in the prefill phase; CoDec is complementary and targets the decode phase, where attention becomes memory-bound and shared-prefix KV reads can be combined.

Prefix-tree decoding and multilevel attention. Several recent works explore decoding-time attention on prefix trees. ChunkAttention [56] and DeFT [53] consider tree-structured inference and propose prefix-aware attention execution strategies. FlashInfer proposes multilevel cascade attention for shared-prefix batch decoding [57]. CoDec differs from these approaches in two key aspects: (1) it performs workload partitioning with a global view of the entire prefix tree to mitigate

imbalance, rather than treating each prefix node independently; and (2) it uses a parallel, tree-structured reduction strategy to aggregate partial results efficiently, avoiding the overhead of launching many reduction kernels when the tree contains a large number of nodes.

Other attention mechanisms. In addition to multi-head attention (MHA), Multi-query attention (MQA) [38], Grouped-query attention (GQA) [5], other advanced attention mechanisms have been proposed. [10, 11] propose multi-head latent attention (MLA), a novel paradigm that projects queries, keys, and values into a low-dimensional latent space across multiple heads. CoDec can support MHA/MQA/GQA directly and can be extended to MLA by reconstructing per-head KV blocks before applying the same prefix-aware attention and reduction pipeline.

Distributed KV cache management. PagedAttention [22] integrates the paged memory management mechanism into attention computation, mitigating memory fragmentation and enhancing inference throughput. [20, 34] explore a distributed environment, employing a memory pool for KV caching across multiple instances. Specifically, [34] utilizes hashing, while [20] adopts a global prompt tree, to retrieve historical KV cache. We notice that CoDec is orthogonal to these approaches, as it considers the KV cache of runtime attention computation on a single instance, and can be combined with these storage-level optimizations.

Distributed attention computation. With the rapid growth of model sizes and sequence lengths, the need for distributed attention computation has become increasingly important. When employing traditional parallelization methods to distribute attention computation, tensor parallelism [39] is employed on the head dimension, while data parallelism [33] partitions the batch dimension. Recently, for scenarios involving long sequence lengths, sequence parallelism [24, 49, 52], involves partitioning along the sequence dimension. CoDec can be easily integrated with tensor parallelism as head dimension do not affect our design, while the sequence parallelism and data parallelism may lead to a lower sharing ratio, which is an interesting direction to explore the task division in these distributed settings.

9 Conclusion

In this paper, we presented CoDec, a dedicated prefix-shared decoding operator designed to significantly accelerate attention computation, which dominates the memory-bound LLM decode stage as the primary performance bottleneck, by efficiently leveraging shared KV cache patterns across multiple requests. Our approach introduces two key innovations: (1) a novel shared-prefix attention kernel that optimizes memory hierarchy through sophisticated indexing between prefix KV cache trees and query tensors while exploiting both intra-block and inter-block parallelism; and (2) a comprehensive workload balancing mechanism featuring a profile-based cost estimator, intelligent task division, and efficient scheduling algorithms to handle irregular workloads. Experimental results show that CoDec achieves significant performance improvements over state-of-the-art FlashDecoding kernels, with up to 11.56× speedup and 150.56× memory access reduction across diverse workloads.

Acknowledgments

This work was supported by the Key Program of the National Natural Science Foundation of China under Grant Nos. 62325205, 62502193 and 62272215, the Natural Science Foundation of Jiangsu Province under Grant Nos. BK20243053, Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (No. JYB2025XDXM901), the Nanjing "U35" Talent Cultivation Program (No. U (2024) 001), Nanjing Kunpeng&Ascend Center of Cultivation, and Nanjing University-China Mobile Communications Group Co., Ltd. Joint Institute. Rong Gu is the corresponding author.

References

[1] 2020. NVIDIA A100 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/a100/. [Accessed 15-04-2025].

[2] 2024. meta-llama/Llama-3.1-8B. https://huggingface.co/meta-llama/Llama-3.1-8B. [Accessed 15-04-2025].

[3] Shubham Agarwal, Sai Sundaresan, Subrata Mitra, Debabrata Mahapatra, Archit Gupta, Rounak Sharma, Nirmal Joshua Kapu, Tong Yu, and Shiv Saini. 2025. Cache-Craft: Managing Chunk-Caches for Efficient Retrieval-Augmented Generation. *Proc. ACM Manag. Data* 3, 3, Article 136 (June 2025), 28 pages. doi:10.1145/3725273

[4] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, and Ramachandran Ramjee. 2023. SARATHI: Efficient LLM Inference by Piggybacking Decodes with Chunked Prefills. arXiv:2308.16369 [cs.LG] https://arxiv.org/abs/2308.16369

[5] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. arXiv:2305.13245 [cs.CL] https://arxiv.org/abs/2305.13245

[6] 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. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL] https://arxiv.org/abs/2005.14165

[7] Tri Dao. 2024. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In *International Conference on Learning Representations (ICLR)*.

[8] Tri Dao, Daniel Haziza, Francisco Massa, and Grigory Sizov. 2023. Flash-Decoding for long-context inference. https://crfm.stanford.edu/2023/10/12/flashdecoding.html. [Accessed 08-04-2025].

[9] DeepSeek. 2024. DeepSeek-R1-Lite-Preview is now live: unleashing supercharged reasoning power. https://api-docs.deepseek.com/news/news1120.

[10] DeepSeek-AI, Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao Yang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jin Chen, Jingyang Yuan, Junjie Qiu, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruizhe Pan, Runxin Xu, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Size Zheng, T. Wang, Tian Pei, Tian Yuan, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Liu, Xin Xie, Xingkai Yu, Xinnan Song, Xinyi Zhou, Xinyu Yang, Xuan Lu, Xuecheng Su, Y. Wu, Y. K. Li, Y. X. Wei, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Zheng, Yichao Zhang, Yiliang Xiong, Yilong Zhao, Ying He, Ying Tang, Yishi Piao, Yixin Dong, Yixuan Tan, Yiyuan Liu, Yongji Wang, Yongqiang Guo, Yuchen Zhu, Yuduan Wang, Yuheng Zou, Yukun Zha, Yunxian Ma, Yuting Yan, Yuxiang You, Yuxuan Liu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhewen Hao, Zhihong Shao, Zhiniu Wen, Zhipeng Xu, Zhongyu Zhang, Zhuoshu Li, Zihan Wang, Zihui Gu, Zilin Li, and Ziwei Xie. 2024. DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model. arXiv:2405.04434 [cs.CL] https://arxiv.org/abs/2405.04434

[11] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song,

Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. 2025. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] https://arxiv.org/abs/2412.19437

[12] Dom Eccleston. 2023. ShareGPT. https://github.com/domeccleston/sharegpt.

[13] Chao Fang, Man Shi, Robin Geens, Arne Symons, Zhongfeng Wang, and Marian Verhelst. 2025. Anda: Unlocking efficient LLM inference with a variable-length grouped activation data format. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1467–1481.

[14] Yichao Fu, Peter Bailis, Ion Stoica, and Hao Zhang. 2024. Break the sequential dependency of llm inference using lookahead decoding. *arXiv preprint arXiv:2402.02057* (2024).

[15] Github. 2024. Accelerate your development speed with copilot. https://copilot.github.com.

[16] R. L. Graham. 1966. Bounds for certain multiprocessing anomalies. *The Bell System Technical Journal* 45, 9 (1966), 1563–1581. doi:10.1002/j.1538-7305.1966.tb01709.x

[17] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[18] Zhicheng Guo, Sijie Cheng, Hao Wang, Shihao Liang, Yujia Qin, Peng Li, Zhiyuan Liu, Maosong Sun, and Yang Liu. 2024. StableToolBench: Towards Stable Large-Scale Benchmarking on Tool Learning of Large Language Models. arXiv:2403.07714 [cs.CL] https://arxiv.org/abs/2403.07714

[19] Yupeng Hou, Junjie Zhang, Zihan Lin, Hongyu Lu, Ruobing Xie, Julian McAuley, and Wayne Xin Zhao. 2023. Large Language Models are Zero-Shot Rankers for Recommender Systems. *ArXiv* abs/2305.08845 (2023). https://api.semanticscholar.org/CorpusID:258686540

[20] Cunchen Hu, Heyang Huang, Junhao Hu, Jiang Xu, Xusheng Chen, Tao Xie, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, and Yizhou Shan. 2024. MemServe: Context Caching for Disaggregated LLM Serving with Elastic Memory Pool. arXiv:2406.17565 [cs.DC] https://arxiv.org/abs/2406.17565

[21] Ehsan Kamalloo, Nouha Dziri, Charles Clarke, and Davood Rafiei. 2023. Evaluating Open-Domain Question Answering in the Era of Large Language Models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (Eds.). Association for Computational Linguistics, Toronto, Canada, 5591–5606. doi:10.18653/v1/2023.acl-long.307

[22] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. doi:10.1145/3600006.3613165

[23] Jiaqi Li, Mengmeng Wang, Zilong Zheng, and Muhan Zhang. 2024. LooGLE: Can Long-Context Language Models Understand Long Contexts? arXiv:2311.04939 [cs.CL] https://arxiv.org/abs/2311.04939

[24] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. 2022. Sequence Parallelism: Long Sequence Training from System Perspective. arXiv:2105.13120 [cs.LG] https://arxiv.org/abs/2105.13120

[25] Yuhang Li, Rong Gu, Chengying Huan, Zhibin Wang, Renjie Yao, Chen Tian, and Guihai Chen. 2025. HotPrefix: Hotness-Aware KV Cache Scheduling for Efficient Prefix Sharing in LLM Inference Systems. *Proc. ACM Manag. Data* 3, 4, Article 250 (Sept. 2025), 27 pages. doi:10.1145/3749168

[26] Zachary C. Lipton, John Berkowitz, and Charles Elkan. 2015. A Critical Review of Recurrent Neural Networks for Sequence Learning. arXiv:1506.00019 [cs.LG] https://arxiv.org/abs/1506.00019

[27] Junyu Luo, Weizhi Zhang, Ye Yuan, Yusheng Zhao, Junwei Yang, Yiyang Gu, Bohan Wu, Binqi Chen, Ziyue Qiao, Qingqing Long, Rongcheng Tu, Xiao Luo, Wei Ju, Zhiping Xiao, Yifan Wang, Meng Xiao, Chenwu Liu, Jingyang Yuan, Shichang Zhang, Yiqiao Jin, Fan Zhang, Xian Wu, Hanqing Zhao, Dacheng Tao, Philip S. Yu, and Ming Zhang. 2025. Large Language Model Agent: A Survey on Methodology, Applications and Challenges. arXiv:2503.21460 [cs.CL] https://arxiv.org/abs/2503.21460

[28] Shaobo Ma, Chao Fang, Haikuo Shao, and Zhongfeng Wang. 2025. APT-LLM: Exploiting Arbitrary-Precision Tensor Core Computing for LLM Acceleration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2025).

[29] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. 2024. 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* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 932–949. doi:10.1145/3620666.3651335

[30] Hyungjun Oh, Kihong Kim, Jaemin Kim, Sungkyun Kim, Junyeol Lee, Du-seong Chang, and Jiwon Seo. 2024. Exegpt: Constraint-aware resource scheduling for llm inference. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 369–384.

[31] OpenAI. 2024. ChatGPT. https://chat.openai.com. Accessed: 2024-08-09.

[32] OpenAI. 2024. Learning to reason with LLMs. https://openai.com/index/learning-to-reason-with-llms/.

[33] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2022. Efficiently Scaling Transformer Inference. arXiv:2211.05102 [cs.LG] https://arxiv.org/abs/2211.05102

[34] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2024. Mooncake: A KVCache-centric Disaggregated Architecture for LLM Serving. arXiv:2407.00079 [cs.DC] https://arxiv.org/abs/2407.00079

[35] QWen. 2024. QwQ: Reflect Deeply on the Boundaries of the Unknown. https://qwenlm.github.io/blog/qwq-32b-preview/.

[36] Laria Reynolds and Kyle McDonell. 2021. Prompt programming for large language models: Beyond the few-shot paradigm. In *Extended abstracts of the 2021 CHI conference on human factors in computing systems*. 1–7.

[37] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[38] Noam Shazeer. 2019. Fast Transformer Decoding: One Write-Head is All You Need. arXiv:1911.02150 [cs.NE] https://arxiv.org/abs/1911.02150

[39] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv:1909.08053 [cs.CL] https://arxiv.org/abs/1909.08053

[40] Significant-Gravitas. 2023. AutoGPT: Build, Deploy, and Run AI Agents. https://github.com/Significant-Gravitas/AutoGPT.

[41] Chan Hee Song, Jiaman Wu, Clayton Washington, Brian M. Sadler, Wei-Lun Chao, and Yu Su. 2023. LLM-Planner: Few-Shot Grounded Planning for Embodied Agents with Large Language Models. arXiv:2212.04088 [cs.AI] https://arxiv.org/abs/2212.04088

[42] Vikranth Srivatsa, Zijian He, Reyna Abhyankar, Dongming Li, and Yiying Zhang. 2024. Preble: Efficient Distributed Prompt Scheduling for LLM Serving. arXiv:2407.00023 [cs.DC] https://arxiv.org/abs/2407.00023

[43] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, et al. 2024. Gemini: A Family of Highly Capable Multimodal Models. arXiv:2312.11805 [cs.CL] https://arxiv.org/abs/2312.11805

[44] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[45] vLLM Project. 2024. vLLM Automatic Prefix Caching. https://docs.vllm.ai/en/latest/features/automatic_prefix_caching.html. Accessed: 2024-08-09.

[46] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. arXiv:2203.11171 [cs.CL] https://arxiv.org/abs/2203.11171

[47] Zhibin Wang, Shipeng Li, Xue Li, Yuhang Zhou, Zhonghui Zhang, Zibo Wang, Rong Gu, Chen Tian, Kun Yang, and Sheng Zhong. 2025. Echo: Efficient Co-Scheduling of Hybrid Online-Offline Tasks for Large Language Model Serving. arXiv:2504.03651 [cs.DC] https://arxiv.org/abs/2504.03651

[48] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (April 2009), 65–76. doi:10.1145/1498765.1498785

[49] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. 2024. LoongServe: Efficiently Serving Long-Context Large Language Models with Elastic Sequence Parallelism. arXiv:2404.09526 [cs.DC] https://arxiv.org/abs/2404.09526

[50] Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. 2024. Inference Scaling Laws: An Empirical Analysis of Compute-Optimal Inference for Problem-Solving with Language Models. arXiv:2408.00724 [cs.AI] https://arxiv.org/abs/2408.00724

[51] Junbin Xiao, Xindi Shang, Angela Yao, and Tat-Seng Chua. 2021. NExT-QA:Next Phase of Question-Answering to Explaining Temporal Actions. arXiv:2105.08276 [cs.CV] https://arxiv.org/abs/2105.08276

[52] Amy Yang, Jingyi Yang, Aya Ibrahim, Xinfeng Xie, Bangsheng Tang, Grigory Sizov, Jeremy Reizenstein, Jongsoo Park, and Jianyu Huang. 2024. Context Parallelism for Scalable Million-Token Inference. arXiv:2411.01783 [cs.DC] https://arxiv.org/abs/2411.01783

[53] Jinwei Yao, Kaiqi Chen, Kexun Zhang, Jiaxuan You, Binhang Yuan, Zeke Wang, and Tao Lin. 2025. DeFT: Decoding with Flash Tree-attention for Efficient Tree-structured LLM Inference. arXiv:2404.00242 [cs.CL] https://arxiv.org/abs/2404.00242

[54] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. 2025. CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion. arXiv:2405.16444 [cs.LG] https://arxiv.org/abs/2405.16444

[55] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. arXiv:2305.10601 [cs.CL] https://arxiv.org/abs/2305.10601

[56] Lu Ye, Ze Tao, Yong Huang, and Yang Li. 2024. ChunkAttention: Efficient Self-Attention with Prefix-Aware KV Cache and Two-Phase Partition. arXiv:2402.15220 [cs.LG] https://arxiv.org/abs/2402.15220

[57] Zihao Ye, Ruihang Lai, Bo-Ru Lu, Chien-Yu Lin, Size Zheng, Lequn Chen, Tianqi Chen, and Luis Ceze. 2024. Cascade Inference: Memory Bandwidth Efficient Shared Prefix Batch Decoding. https://flashinfer.ai/2024/02/02/cascade-inference.html

[58] Yilong Zhao, Shuo Yang, Kan Zhu, Lianmin Zheng, Baris Kasikci, Yang Zhou, Jiarong Xing, and Ion Stoica. 2024. BlendServe: Optimizing Offline Inference for Auto-regressive Large Models with Resource-aware Batching. arXiv:2411.16102 [cs.LG] https://arxiv.org/abs/2411.16102

[59] Zihuai Zhao, Wenqi Fan, Jiatong Li, Yunqing Liu, Xiaowei Mei, Yiqi Wang, Zhen Wen, Fei Wang, Xiangyu Zhao, Jiliang Tang, and Qing Li. 2024. Recommender Systems in the Era of Large Language Models (LLMs). arXiv:2307.02046 [cs.IR] https://arxiv.org/abs/2307.02046

[60] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. arXiv:2312.07104 [cs.AI] https://arxiv.org/abs/2312.07104

[61] Zhen Zheng, Xin Ji, Taosong Fang, Fanghao Zhou, Chuanjie Liu, and Gang Peng. 2024. BatchLLM: Optimizing Large Batched LLM Inference with Global Prefix Sharing and Throughput-oriented Token Batching. arXiv:2412.03594 [cs.CL] https://arxiv.org/abs/2412.03594

[62] Yuhang Zhou, Zhibin Wang, Guyue Liu, Shipeng Li, Xi Lin, Zibo Wang, Yongzhong Wang, Fuchun Wei, Jingyi Zhang, Zhiheng Hu, Yanlin Liu, Chunsheng Li, Ziyang Zhang, Yaoyuan Wang, Bin Zhou, Wanchun Dou, Guihai Chen, and Chen Tian. 2025. Squeezing Operator Performance Potential for the Ascend Architecture. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 1156–1171. doi:10.1145/3676641.3716243

[63] Zixuan Zhou, Xuefei Ning, Ke Hong, Tianyu Fu, Jiaming Xu, Shiyao Li, Yuming Lou, Luning Wang, Zhihang Yuan, Xiuhong Li, Shengen Yan, Guohao Dai, Xiao-Ping Zhang, Yuhan Dong, and Yu Wang. 2024. A Survey on Efficient Inference for Large Language Models. arXiv:2404.14294 [cs.CL] https://arxiv.org/abs/2404.14294

Received October 2025; revised January 2026; accepted February 2026