FlashForge: Ultra-Efficient Prefix-Aware Attention for LLM Decoding

Zhibin Wang[1*], Rui Ning[1*], Chao Fang[1], Zhonghui Zhang[1], Xi Lin[1], Shaobo Ma[1], Mo Zhou[1],
Xue Li[2], Zhongfeng Wang[1], Chengying Huan[1], Rong Gu[1], Kun Yang[1], Guihai Chen[1], Sheng Zhong[1],
Chen Tian[1]

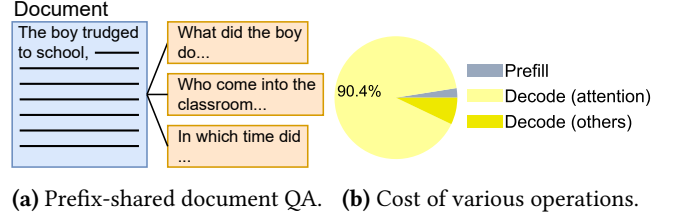[1]*Nanjing University*   [2]*Alibaba Group*

Abstract

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 FlashForge. Flash-Forge delivers two key innovations: a novel shared-prefix attention kernel that optimizes memory hierarchy and 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 FlashForge 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.

Keywords: GPU kernel, large language models (LLMs), attention computation, prefix-sharing

1 Introduction

Large language models (LLMs) have demonstrated significant performance across diverse tasks, such as question answering [19], planning [36, 37], code generation [13, 33], recommendation systems [17, 54], and even solving complex mathematical problems [8, 28, 31, 46]. Despite their impressive capabilities, inference efficiency remains critical for LLM deployment, as it directly impacts user experience and operational costs [58]. For instance, real-time applications such as chatbots and interactive AI agents require low-latency responses to maintain seamless interactions [24]. Meanwhile, enterprise-scale deployments, where LLMs process millions

---
*Both authors contributed equally to this paper.

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

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

of queries daily, must optimize computational resources to remain cost-effective [7, 20].

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 [11, 21, 47], tree-of-thoughts [50], speculative decoding [25], and few-shot prompting [32]. For example, as shown in Figure 1 (a), in document-based question answering, multiple questions may pertain to the same document [11, 21, 47]. 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 [30, 38, 41, 43, 53, 55, 56]. 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 CodeLlama-13B [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 [12]. Recently, state-of-the-art attention optimization methods, such as FlashAttention [6] and FlashDecoding [7], have achieved significant speedup by leveraging the shared memory (i.e., on-chip memory) to reduce memory access overhead and increasing the parallelism of the attention computation. Consider a batch of multi-head attention that takes three 4D tensors as input: the query ($Q \in \mathbb{R}^{bs \times n_q \times h \times d}$),

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/FlashDecoding decomposes the attention operation in *batch, head, query sequence, and KV sequence dimensions* into multiple blocks, to ensure that each block can fit in shared memory and support 4-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.

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 [6, 7] 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 [49, 55], 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 [51, 52] 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 [26]. Moreover, varying shapes of query tensors and prefix KV cache nodes result in divergent memory or compute bounds [44, 57], 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.
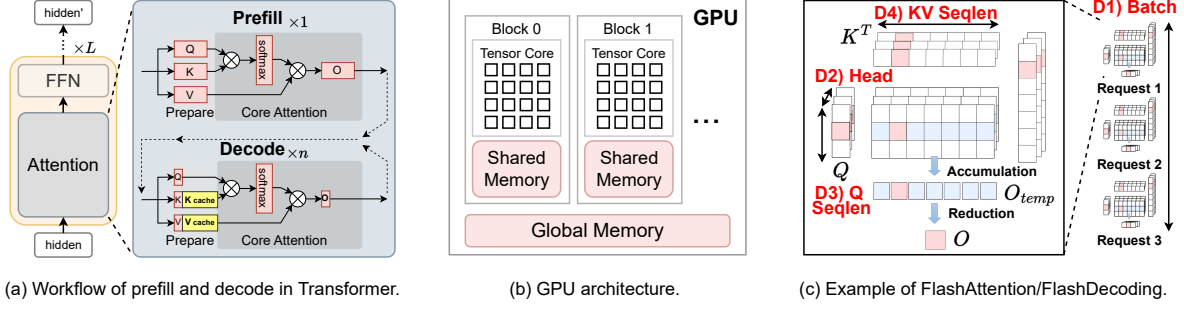
To address the above challenges, we develop the dedicated prefix-shared attention operator, namely FlashForge, which combines the memory access of the attention computation of the shared prefix across different requests in the decode stage. FlashForge delivers two key innovations: First, it implements 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. 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.

Extensive evaluation (Section 6). We conduct extensive experiments considering various workloads to demonstrate the effectiveness of FlashForge in terms of speedup and memory access reduction. Compared to the state-of-the-art FlashDecoding, FlashForge achieves an average speedup of 1.9× and memory access reduction of 120.9× for attention computation, and 3.8× for the end-to-end latency.

---

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

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

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

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 Background

2.1 LLM Inference

Transformer architecture: Recent mainstream LLMs, such as ChatGPT [27], DeepSeek [10], Llama [15], and Gemini [39] are based on the transformer architecture [40], 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 [23] considering the long-range dependencies [40]. The FFN module is responsible for learning complex representations of the tokens. These two modules are typically stacked L times to deepen the model.

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 and decode time proportion of two workloads when serving the Llama3.1-8B model. 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.

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, the self-attention operation takes a sequence of input tokens, 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 will be generated from the input tokens' embeddings through three linear transformations, 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 leveraging chunked-refill [3].

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

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:

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

m is the maximum value of the vector $\mathbf{x}$ to prevent overflow in the exponentiation operation [6].

In this paper, we use multi-head attention [40] as the default attention mechanism, and other attention mechanisms [4, 9, 34] will be discussed in Section 7. 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 ($Q \in \mathbb{R}^{bs \times n_q \times h \times d}$), key ($K \in \mathbb{R}^{bs \times n \times h \times d}$), and value ($V \in \mathbb{R}^{bs \times n \times h \times d}$) 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) [44, 57] 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 [6] and its successor FlashDecoding [7] 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.

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

Algorithm 1 FlashAttention

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

2.5 Prefix-sharing

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

- **Document Question Answering (QA) [11, 21, 47].** Users may ask multiple questions about the same document. For instance, in the LooGLE [21] dataset, the average prompt length is 23474 tokens, and the sharing rate is 91%.
- **Tool-use [16].** 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 [16] has an average prompt length of 1835 tokens and a sharing rate of 85%.
- **Few-shot Prompting [5].** This technique often involves prepending identical instructions or examples (e.g., demonstrations of tool usage) to various distinct prompts.
- **Self-consistency [42].** 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 [50].** 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 [25].** 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 [18, 30, 43, 53, 55, 56] 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.*

2.6 Issues

The aforementioned limitations highlight two critical issues to be urgently addressed for efficient GPU-based prefix-shared decoding.

Issue 1: From regular 4D tensors to irregular tree of 3D tensors makes the parallelization more complex. In FlashAttention/FlashDecoding, KV tensors utilize 4D structures that: 1) efficiently materialize in global memory, 2) divide naturally into parallel processing blocks, and 3) support straightforward reduction operations across KV sequence dimensions. Conversely, prefix-shared decoding transforms these into 3D tensors, which: 1) complicates KV cache and query tensor indexing, 2) increases task division complexity, and 3) necessitates tree-based reduction operations for result merging. These structural changes demand innovative tensor organization approaches to maintain computational efficiency despite the irregular data structures.

Issue 2: Varying KV sequence length and degree of sharing in each KV cache node results in significantly different workloads. For request groups sharing a prefix in a KV cache node, both KV cache sequence length varies according to prefix length, and query tensor sequence length fluctuates based on the sharing group size. This variable workload complicates load balancing across GPU blocks, often causing resource under-utilization through processing stalls. The non-linear relationship between workload characteristics and performance further challenges effective task division and scheduling. While fine-grained partitioning improves workload balance, it introduces substantial scheduling and reduction overhead. Conversely, coarse-grained approaches suffer from persistent load imbalances. This drives us to develop adaptive scheduling algorithms that dynamically optimize resource utilization while minimizing overhead costs.

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.
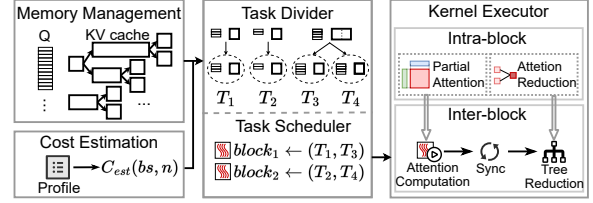

Figure 3. Overview of FlashForge.

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 also maintain the index between the query tensor and the KV cache tree, thereby virtually constructing the view of each partial attention computation.

Kernel Executor (Section 4.2, 4.3): The kernel executor considers two levels of parallelism: intra-block and inter-block. We abstract two critical intra-block kernel primitives from FlashAttention and FlashDecoding, including the partial attention computation kernel and the partial output reduction kernel. Upon these two intra-block kernel primitives, we develop the inter-block kernel executor, which composes the following steps: Firstly, the kernel executor will launch the partial attention computation kernel for each computation block. After all computations are finished, i.e., synchronization, the kernel executor will conduct the tree reduction to merge the results for each query by leveraging the partial output reduction kernel.

Cost Estimator (Section 5.2): The cost estimator is responsible for estimating the cost of *a block execution the partial attention computation kernel with the given query tensor and the KV cache tensor.* For various workloads, specifically, the shape of the query tensor and the KV cache tensor, the kernel execution can be either compute-bound or memory-bound, which makes theoretical performance estimation intractable. Moreover, hardware configuration will also affect the performance of the kernel execution. Therefore, after given the hardware configuration and deployed model, the cost estimator will conduct several micro-benchmarks to profile the performance of the kernel execution with different workloads. For the unprofiled workloads, the cost estimator will leverage the interpolation to estimate the performance.
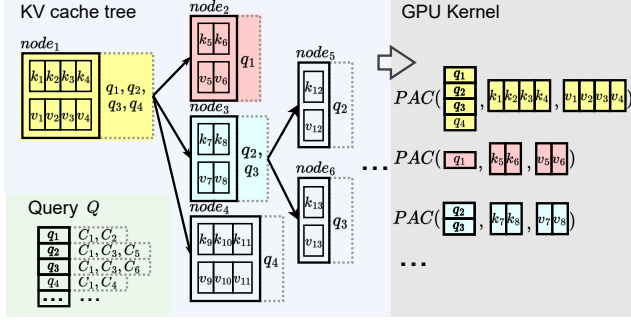
Figure 4. KV cache forest.

Task Divider and Scheduler (Section 5.1): Given a batch of decoding requests, the partial attention computation workload among different KV cache nodes and its corresponding query tensor varies significantly. The task divider and scheduler will divide the partial attention computation workload into several subtasks, and determine the execution order of these subtasks in each block.

4 GPU Kernel Design

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.

4.1 KV Cache Management

Different from the traditional KV cache management systems, e.g., PagedAttention [20] 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.

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

Algorithm 2 PAC	**Algorithm 3** POR
1: **Input:** Q, K, V	1: **Input:** $O_1, O_2, m_1, m_2, s_1, s_2$
2: **Output:** O	2: **Output:** O
3: $S \leftarrow Q \cdot K^T / \sqrt{d}$	3: $m \leftarrow \max(m_1, m_2)$
4: $S \leftarrow \text{SOFTMAX}(S)$	4: $s \leftarrow s_1 e^{m_1 - m} + s_2 \cdot e^{m_2 - m}$
5: $O \leftarrow S \cdot V$	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	6: **return** O

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.

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.

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.

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.

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 FlashForge

1: **Input:** Q, (K, V)
2: **Output:** O
3: Initialize $\mathbf{O}_{tree}$ with the tree structure of (K, V)
4: **for** $(K, V) \in$ (K, 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** O

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 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 FlashForge 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 FlashForge is about $\overline{n_q}$ times lower than that of FlashAttention. Regarding the computation complexity, FlashForge is the same as FlashAttention.

5 Workload Balance

The workload of the partial attention computation between each KV cache node and its corresponding queries can be varied significantly as **the length of the KV cache node varies** and **the number of queries varies**. For example, in tree-of-thoughts tasks [50], different nodes may have varying lengths, and different branches may have varying depths and numbers of parallel explorations. Therefore, straightforward launching of the PAC kernel for each KV cache node and its corresponding queries as shown in Figure 4 suffers from the workload imbalance resulting in under-utilization of GPU resources.

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:

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

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 [14], 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 $O(n_q \times n \times d)$. Moreover, the global memory access is the sum of the memory access of Q, K and V, which is $O((n_q + 2n) \times d)$. 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.

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

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.

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 Evaluation

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

- **Section 6.1: What is the benefit of using Flash-Forge?** We compare FlashForge with the SOTA attention kernel, FlashAttention [6] and FlashDecoding [7], in terms of attention operation time, end-to-end time, and global memory IO.
- **Section 6.2: How does each optimization contribute to the performance?** We conduct an ablation study to analyze the contribution of each optimization in FlashForge.

- **Section 6.3: How do the specific design choices impact FlashForge's performance?** We analyze the impact of key design decisions in FlashForge, particularly focusing on the optimal division granularity for different sequence lengths.
- **Section 6.4: How does FlashForge perform on different GPUs?** We evaluate the performance of FlashForge on different GPUs, including NVIDIA H800, A100, RTX 4090, A30, and RTX A6000.

FlashForge is implemented in around 2500 lines of CUDA and Python codes. Unless otherwise specified, the results are averaged over 3 runs, and experiments are conducted 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. By default, we use the CodeLlama-13B model from Meta, which configures 32 query attention heads, key and value heads, and a head dimension of 128.

6.1 Comparison with SOTA

To further reveal the performance characteristics of Flash-Forge, we conduct a series of experiments to evaluate the impact of different workloads on the performance of Flash-Forge. 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 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 FlashForge in terms of 1) attention kernel execution time, 2) global memory access in 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 Flash-Forge with FlashDecoding [6] provided by FlashAttention 2.7.4, which is the SOTA attention kernel for long-context decoding. For end-to-end latency, we integrate FlashForge into PyTorch 2.6.0 and manage KV cache node in each attention layer, and compare it with vLLM 0.6.6 [20].

Attention Execution Time. As shown in Figure 7, Flash-Forge 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 FlashForge 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 FlashForge 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 FlashForge and FlashDecoding, which verifies the performance gain of FlashForge. The global memory access of FlashForge 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 7, which is attributed to the workload balance and scheduling strategy.

End-to-End Latency. We also evaluate the end-to-end latency of FlashForge 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 APPS 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 FlashForge 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.

6.2 Ablation Studies

To better understand how each innovation contributes to the overall performance gains, we conduct an ablation study analyzing three key optimizations: (1) KV cache access combination (shared prefix tree), (2) proper partitioning of workload,
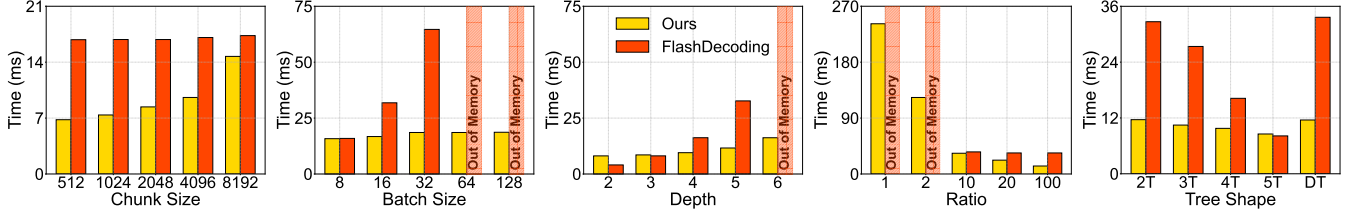
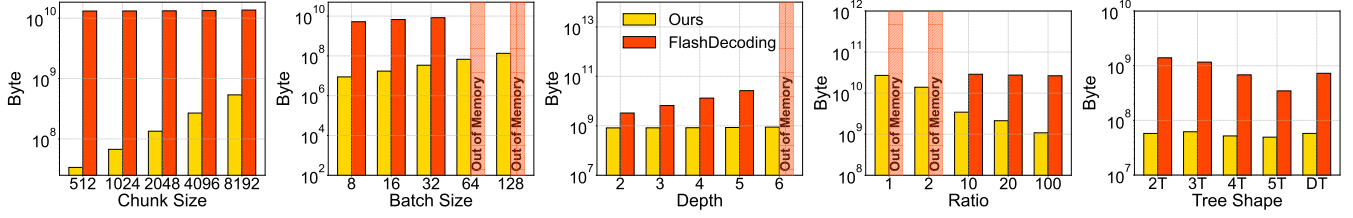**Figure 5.** FlashForge vs. FlashDecoding on execution time.

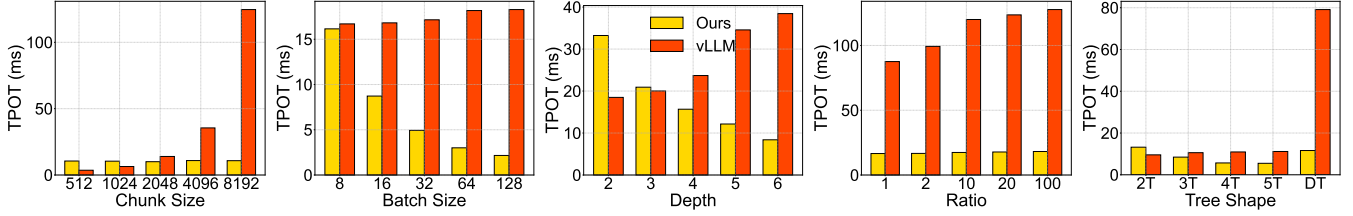**Figure 6.** FlashForge vs. FlashDecoding on global memory access.

Figure 7. FlashForge vs. vLLM on end-to-end time.

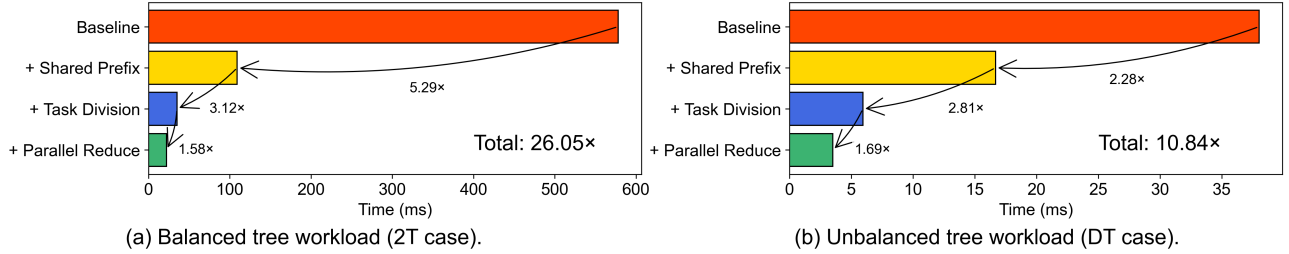(a) Balanced tree workload (2T case). (b) Unbalanced tree workload (DT case).

Figure 8. Ablation Study.

and (3) parallel execution of combine kernels. For comprehensive evaluation, we use a full binary shared prefix tree as our balanced tree workload, and a degenerate tree as our unbalanced tree workload, with a maximum context length of 200k tokens.

The results are shown in Figure 8(a) and Figure 8(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 significantly to reducing latency, with the combination of all three providing the maximum speedup. Notably, the impact of workload balancing and parallelism is more significant for the balanced tree, due to its higher intrinsic computational load.

6.3 Impact of Division Granularity

To further reveal the impact of the granularity of task division, we further consider a naive approach that simply and equally divides each task into a fixed number of subtasks. This naive approach does not consider the workload distribution of the KV cache tree and query tensor, which can lead to an unbalanced workload among different subtasks or too fine-grained division with high overhead.
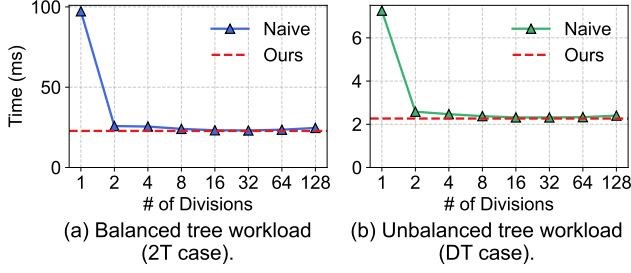
Figure 9. Impact of division granularity.

Figure 9 shows the performance of the naive approach with different number of divisions in comparison with our scheduling strategy. We represent our approach with a horizontal dashed line as it automatically determines the optimal number of divisions based on workload distribution. When setting the number of divisions to 1, the naive approach is equivalent to the baseline implementation, which processes the KV cache tree and query tensor without any division.

The experimental results show that our approach outperforms the best division strategy of the naive approach by 1.02-1.04× and averages 3.80× speedup across all workloads. Compared to the naive approach without division, our approach achieves 3.20-4.39× and averages 3.80× speedup 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.

### 6.4 Performance in Varying GPU Cards

To evaluate the cross-platform efficiency and performance consistency of our method, we conduct experiments with a context length of 50K tokens, representing an extreme long-sequence scenario on five modern GPUs.

As shown in Figure 10, FlashForge consistently outperforms FlashDecoding [7] 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, FlashForge 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 FlashForge is less sensitive to hardware limitations, making it more suitable for deployment across the entire GPU spectrum from data center-oriented to comsumer-grade devices.

7 Related Work

Other attention mechanisms. In addition to multi-head attention (MHA), other advanced attention mechanisms have been proposed. Multi-query attention (MQA) [34] shares a
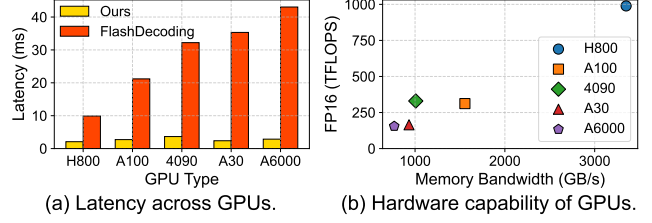
Figure 10. Performance on diverse GPUs.

single key-value pair across all heads' queries, thereby reducing memory usage but sacrificing the model's expressiveness. Grouped-query attention (GQA) [4] strikes a balance by grouping heads to share key-value pairs, offering better modeling capacity than MQA while being more efficient than full MHA. [9, 10] propose multi-head latent attention (MLA), a novel paradigm that projects queries, keys, and values into a low-dimensional latent space across multiple heads. Flash-Forge can easily extend to these attention mechanisms, as it is agnostic to the specific attention mechanism used.

Distributed KV cache management. PagedAttention [20] integrates the paged memory management mechanism into attention computation, mitigating memory fragmentation and enhancing inference throughput. [18, 30] explore a distributed environment, employing a memory pool for KV caching across multiple instances. Specifically, [30] utilizes hashing, while [18] adopts a global prompt tree, to retrieve historical KV cache. We notice that FlashForge 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 [35] is employed on the head dimension, while data parallelism [29] partitions the batch dimension. Recently, for scenarios involving long sequence lengths, sequence parallelism [22, 45, 48], involves partitioning along the sequence dimension. FlashForge 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.

8 Conclusion

In this paper, we presented FlashForge, 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 FlashForge 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.

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] Amey Agrawal, Ashish Panwar, et al. 2023. SARATHI: Efficient LLM Inference by Piggybacking Decodes with Chunked Prefills. arXiv:cs.LG/2308.16369 https://arxiv.org/abs/2308.16369

[4] Joshua Ainslie, James Lee-Thorp, et al. 2023. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. arXiv:cs.CL/2305.13245 https://arxiv.org/abs/2305.13245

[5] Tom B. Brown, Benjamin Mann, et al. 2020. Language Models are Few-Shot Learners. arXiv:cs.CL/2005.14165 https://arxiv.org/abs/2005.14165

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

[7] Tri Dao, Daniel Haziza, et al. 2023. Flash-Decoding for long-context inference. https://crfm.stanford.edu/2023/10/12/flashdecoding.html. [Accessed 08-04-2025].

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

[9] DeepSeek-AI, Aixin Liu, et al. 2024. DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model. arXiv:cs.CL/2405.04434 https://arxiv.org/abs/2405.04434

[10] DeepSeek-AI, Aixin Liu, et al. 2025. DeepSeek-V3 Technical Report. arXiv:cs.CL/2412.19437 https://arxiv.org/abs/2412.19437

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

[12] Yichao Fu, Peter Bailis, et al. 2024. Break the sequential dependency of llm inference using lookahead decoding. *arXiv preprint arXiv:2402.02057* (2024).

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

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

[15] Aaron Grattafiori, Abhimanyu Dubey, et al. 2024. The Llama 3 Herd of Models. arXiv:cs.AI/2407.21783 https://arxiv.org/abs/2407.21783

[16] Zhicheng Guo, Sijie Cheng, et al. 2024. StableToolBench: Towards Stable Large-Scale Benchmarking on Tool Learning of Large Language Models. arXiv:cs.CL/2403.07714 https://arxiv.org/abs/2403.07714

[17] Yupeng Hou, Junjie Zhang, et al. 2023. Large Language Models are Zero-Shot Rankers for Recommender Systems. *ArXiv* abs/2305.08845 (2023). https://api.semanticscholar.org/CorpusID:258686540

[18] Cunchen Hu, Heyang Huang, et al. 2024. MemServe: Context Caching for Disaggregated LLM Serving with Elastic Memory Pool. arXiv:cs.DC/2406.17565 https://arxiv.org/abs/2406.17565

[19] Ehsan Kamalloo, Nouha Dziri, et al. 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, et al. (Eds.). Association for Computational Linguistics, Toronto, Canada, 5591–5606. https://doi.org/10.18653/v1/2023.acl-long.307

[20] Woosuk Kwon, Zhuohan Li, et al. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611âĂŞ626. https://doi.org/10.1145/3600006.3613165

[21] Jiaqi Li, Mengmeng Wang, et al. 2024. LooGLE: Can Long-Context Language Models Understand Long Contexts? arXiv:cs.CL/2311.04939 https://arxiv.org/abs/2311.04939

[22] Shenggui Li, Fuzhao Xue, et al. 2022. Sequence Parallelism: Long Sequence Training from System Perspective. arXiv:cs.LG/2105.13120 https://arxiv.org/abs/2105.13120

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

[24] Junyu Luo, Weizhi Zhang, et al. 2025. Large Language Model Agent: A Survey on Methodology, Applications and Challenges. arXiv:cs.CL/2503.21460 https://arxiv.org/abs/2503.21460

[25] Xupeng Miao, Gabriele Oliaro, et al. 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 (ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 932âĂŞ949. https://doi.org/10.1145/3620666.3651335

[26] Hyungjun Oh, Kihong Kim, et al. 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.

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

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

[29] Reiner Pope, Sholto Douglas, et al. 2022. Efficiently Scaling Transformer Inference. arXiv:cs.LG/2211.05102 https://arxiv.org/abs/2211.05102

[30] Ruoyu Qin, Zheming Li, et al. 2024. Mooncake: A KVCache-centric Disaggregated Architecture for LLM Serving. arXiv:cs.DC/2407.00079 https://arxiv.org/abs/2407.00079

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

[32] 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.

[33] Baptiste Roziere, Jonas Gehring, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

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

[35] Mohammad Shoeybi, Mostofa Patwary, et al. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv:cs.CL/1909.08053 https://arxiv.org/abs/1909.08053

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

[37] Chan Hee Song, Jiaman Wu, et al. 2023. LLM-Planner: Few-Shot Grounded Planning for Embodied Agents with Large Language Models. arXiv:cs.AI/2212.04088 https://arxiv.org/abs/2212.04088

[38] Vikranth Srivatsa, Zijian He, et al. 2024. Preble: Efficient Distributed Prompt Scheduling for LLM Serving. arXiv:cs.DC/2407.00023 https:

//arxiv.org/abs/2407.00023

[39] Gemini Team, Rohan Anil, et al. 2024. Gemini: A Family of Highly Capable Multimodal Models. arXiv:cs.CL/2312.11805 https://arxiv.org/abs/2312.11805

[40] Ashish Vaswani, Noam Shazeer, et al. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

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

[42] Xuezhi Wang, Jason Wei, et al. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. arXiv:cs.CL/2203.11171 https://arxiv.org/abs/2203.11171

[43] Zhibin Wang, Shipeng Li, et al. 2025. Echo: Efficient Co-Scheduling of Hybrid Online-Offline Tasks for Large Language Model Serving. arXiv:cs.DC/2504.03651 https://arxiv.org/abs/2504.03651

[44] Samuel Williams, Andrew Waterman, et al. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (April 2009), 65âĂŞ76. https://doi.org/10.1145/1498765.1498785

[45] Bingyang Wu, Shengyu Liu, et al. 2024. LoongServe: Efficiently Serving Long-Context Large Language Models with Elastic Sequence Parallelism. arXiv:cs.DC/2404.09526 https://arxiv.org/abs/2404.09526

[46] Yangzhen Wu, Zhiqing Sun, et al. 2024. Inference Scaling Laws: An Empirical Analysis of Compute-Optimal Inference for Problem-Solving with Language Models. arXiv:cs.AI/2408.00724 https://arxiv.org/abs/2408.00724

[47] Junbin Xiao, Xindi Shang, et al. 2021. NExT-QA:Next Phase of Question-Answering to Explaining Temporal Actions. arXiv:cs.CV/2105.08276 https://arxiv.org/abs/2105.08276

[48] Amy Yang, Jingyi Yang, et al. 2024. Context Parallelism for Scalable Million-Token Inference. arXiv:cs.DC/2411.01783 https://arxiv.org/abs/2411.01783

[49] Jiayi Yao, Hanchen Li, et al. 2025. CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion. arXiv:cs.LG/2405.16444 https://arxiv.org/abs/2405.16444

[50] Shunyu Yao, Dian Yu, et al. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. arXiv:cs.CL/2305.10601 https://arxiv.org/abs/2305.10601

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

[52] Zihao Ye, Ruihang Lai, et al. 2024. Cascade Inference: Memory Bandwidth Efficient Shared Prefix Batch Decoding. https://flashinfer.ai/2024/02/02/cascade-inference.html

[53] Yilong Zhao, Shuo Yang, et al. 2024. BlendServe: Optimizing Offline Inference for Auto-regressive Large Models with Resource-aware Batching. arXiv:cs.LG/2411.16102 https://arxiv.org/abs/2411.16102

[54] Zihuai Zhao, Wenqi Fan, et al. 2024. Recommender Systems in the Era of Large Language Models (LLMs). arXiv:cs.IR/2307.02046 https://arxiv.org/abs/2307.02046

[55] Lianmin Zheng, Liangsheng Yin, et al. 2024. SGLang: Efficient Execution of Structured Language Model Programs. arXiv:cs.AI/2312.07104 https://arxiv.org/abs/2312.07104

[56] Zhen Zheng, Xin Ji, et al. 2024. BatchLLM: Optimizing Large Batched LLM Inference with Global Prefix Sharing and Throughput-oriented Token Batching. arXiv:cs.CL/2412.03594 https://arxiv.org/abs/2412.03594

[57] Yuhang Zhou, Zhibin Wang, et al. 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 (ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 1156âĂŞ1171.

https://doi.org/10.1145/3676641.3716243

[58] Zixuan Zhou, Xuefei Ning, et al. 2024. A Survey on Efficient Inference for Large Language Models. arXiv:cs.CL/2404.14294 https://arxiv.org/abs/2404.14294