BatchLLM: Optimizing Large Batched LLM Inference with Global Prefix Sharing and Throughput-oriented Token Batching

Zhen Zheng
Microsoft
zhengzhen@microsoft.com

Xin Ji
Microsoft
xinji1@microsoft.com

Taosong Fang*
Chinese Information Processing Laboratory, ISCAS
UCAS
fangtaosong2022@iscas.ac.cn

Fanghao Zhou
Microsoft
fanghaozhou@microsoft.com

Chuanjie Liu
Microsoft
chuanjieliu@microsoft.com

Gang Peng
Microsoft
penggang@microsoft.com

FCFS导致一个decode比例更大的req会被更晚调度，进而无法与prefill chunks进行batch

ABSTRACT

Large language models (LLMs) increasingly play an important role in a wide range of information processing and management tasks. Many of these tasks are performed in large batches or even offline, and the performance indictor for which is throughput. These tasks usually show the characteristic of prefix sharing, where different prompt input can partially show the common prefix. However, the existing LLM inference engines tend to optimize the streaming requests and show limitations of supporting the large batched tasks with the prefix sharing characteristic. The existing solutions use the LRU-based cache to reuse the KV context of common prefix between requests. The KV context that are about to be reused may prematurely evicted with the implicit cache management. Even if not evicted, the lifetime of the shared KV context is extended since requests sharing the same context are not scheduled together, resulting in larger memory usage. These streaming oriented systems schedule the requests in the first-come-first-serve or similar order. As a result, the requests with larger ratio of decoding steps may be scheduled too late to be able to mix with the prefill chunks to increase the hardware utilization. Besides, the token and request number based batching can limit the size of token-batch, which keeps the GPU from saturating for the iterations dominated by decoding tokens. We propose BatchLLM to address the above problems. BatchLLM explicitly identifies the common prefixes globally. The requests sharing the same prefix will be scheduled together to reuse the KV context the best, which also shrinks the lifetime of common KV memory. BatchLLM reorders the requests and schedules the requests with larger ratio of decoding first to better mix the decoding tokens with the latter prefill chunks, and applies memory-centric token batching to enlarge the token-batch sizes, which helps to increase the GPU utilization. Finally, BatchLLM optimizes the prefix-shared Attention kernel with horizontal fusion to reduce tail effect and kernel launch overhead. Extensive evaluation shows that BatchLLM outperforms vLLM by 1.1× to 2.0× on a set of microbenchmarks and two typical industry workloads.

1 INTRODUCTION

The modern information processing and management tasks are starting to leverage large language models (LLMs) to achieve better results, such as the search engine [35, 38], advertisement [23], and recommender systems [19, 43]. Due to the huge amount of data to be processed, many of these LLM tasks are performed in large batches or even offline (e.g., offline ranking), for which the most important performance indicator is throughput. Besides, different input (prompts) of these tasks can partially share the same prefix (e.g., the same web document in the search engine tasks). Given the high cost of LLM inference, how to serve these tasks efficiently is a critical problem for the information processing systems.

The recent LLM inference engines [20, 32] can flexibly support the execution of LLM inference tasks, especially with blocked KV memory management [20] and per-iteration token batching [42]. However, the design of existing inference engines tend to optimize streaming online services and show limitations in throughput-oriented large-batch processing. 1) *Lack of global prefix sharing*. The state-of-the-art (SOTA) LLM inference engine [20] supports to cache the KV context with LRU policy to enable prefix sharing between prompts. However, from a global perspective of large batches, an LRU-based cache may prematurely evict KV contexts that are about to be reused and retain unused KV contexts for a long time, causing unnecessary recalculation. 2) *Suboptimal token batching for throughput-oriented and prefix-shared tasks*. The current LLM inference engines schedule different requests independently and do not cluster the requests with common prefix together to schedule. It can extend the lifetime of the common prefix and thus increase the memory usage, and may prematurely evict KV contexts that are about to be reused as discussed above. Besides, they usually schedule the requests in the first-come-first-serve or similar order to guarantee the fairness and latency of the streaming requests. The requests with larger ratio of decoding steps may be scheduled too late to be able to mix with the prefill chunks to increase the hardware utilization. They apply the mix of prefill and decoding tokens together according to the threshold of token number and request number. This limits the number of decoding tokens that can be batched and keeps the GPU[1] from saturating for the iterations dominated by decoding tokens. 3) *Room for improvement in Attention*. The SOTA prefix-shared Attention optimization can be further optimized with horizontal kernel fusion of the computation on different KV chunks, which is effective to reduce the tail effect and kernel launch overhead.

*Work was done when Taosong Fang interned at Microsoft, advised by Zhen.

[1]This paper illustrates the technique with the GPU architecture. But the basic principle can be applied to other architectures.

Given the limitations of existing works, the LLM inference engines face significant demands in supporting emerging throughput-oriented large batched tasks, especially as the LLM is more and more widely used in the huge amounts of batched information processing and management.

In this work, we introduce BatchLLM, the holistic optimization for the throughput-oriented large-batch LLM inference with global prefix preprocessing, prefix-aware and throughput-oriented token batching and Attention kernel optimization. The basic insight is based on the fact that, the prompt characteristics are known ahead before processing the large batch. Compared to the implicit LRU cache that cannot retain the KV context for reuse accurately, the common prefixes between prompts can be identified explicitly and can be reused explicitly. Besides, the length distribution of the prompts is known ahead and the main performance target is throughput, which allows the inference engine to reorder the requests and form larger token-batches[2].

Building upon these insights, this paper proposes a set of optimizations. It builds the global prefix tree on the large batch ahead-of-time to enable the explicit prefix sharing and avoid the problem of premature eviction of KV contexts. To simplify the design and reduce the overhead of token batching and Attention kernel on multi-level prefix, it simplifies the multi-level prefix into a single level one while retaining the similar KV reusing ratio with Dynamic Programming algorithm on the tree. It schedules the requests sharing the common prefix together to enable them reusing the KV context and reduce the lifetime of the common KV. To form large token-batches at each iteration to increase GPU utilization, it reorders the requests ahead-of-time according to the length of prompt and the decoding (estimated). The request with larger ratio of decoding length to prompt length will be scheduled first so that the decoding tokens will be processed earlier and can be mixed with the latter longer prefill chunks. It also batches the tokens according to the KV memory usage rather than only depending on request number threshold, to form large batches for decoding tokens. Finally, it applies horizontal fusion of Attention calculation on different KV chunks to reduce the overhead of tail effect and kernel launch of the basic prefix-shared Attention implementation.

We have implemented BatchLLM on top of vLLM [20] and conducted extensive evaluations using a set of microbenchmarks of different datasets and models and two industry workloads on NVIDIA and AMD GPUs. BatchLLM achieves end-to-end performance improvement of 1.1× to 2.0× than vLLM for the microbenchmarks and the two industry workloads. The performance gain comes from BatchLLM's ability to reuse the common prefix explicit in the global view, form the token-batches with higher compute-intensity (the ratio of computational operations to memory size accessed), and the more efficient prefix-shared Attention.

In summary, this paper makes the following contributions:

► It pioneers the LLM inference optimization for the throughput oriented large batched prefix-shared scenarios with the insight of using global information of the batch, for the increasingly important LLM-based information processing and management.

► It designs the ahead-of-time prefix identification and enlargement method to achieve effective prefix reusing in global view, proposes the requests reordering and memory-centric token batching method to increase the per-iteration GPU utilization, and implements the horizontal fused kernel of prefix-shared Attention.

► It presents BatchLLM implementation building upon vLLM. Extensive evaluation demonstrates the efficacy of BatchLLM.

[2]This paper uses the term *token-batch* to represent a batch of tokens to be processed for an iteration of continuous batching (background at Sec.2.3). It differs from the concept of *large batch* of all the prompt to be processed of a job.

2 LLM INFERENCE BACKGROUND

2.1 LLM Inference and Its Performance Factors

The auto-regressive LLM inference usually consists of two phases: the *prefill* phase to process the input prompt, and the *decoding* phase to generate the output.

The prefill phase is usually compute-intensive and bounded by the massive computation. This is because the input tokens are all known ahead, and the computation of these tokens can be performed in parallel. The decoding phase is performed differently from the prefill phase: the output tokens are generated one by one and cannot be performed in parallel[3]. As a result, the decoding phase can be easily bounded by the memory access of the weight due to the memory wall problem [36, 37]. Enlarging the batch size can help to push the decoding computation to compute-bound, but the batch size may be limited by the GPU memory capacity. In real practice, the batching of the decoding can be suboptimal due to either the lack of memory space or the inefficient token batching, leading to low hardware utilization.

Attention can be an important performance factor when the sequence is long. According to its algorithm pattern, the basic Attention computation in decoding phase is the matrix-vector multiplication between the (batched) vector Q and matrix KV containing all history context, as tokens from different requests have different KV context. Due to this characteristic, enlarging the batch size usually can not increase the compute utilization of the GPU as it does not increase the compute-intensity.

2.2 KV Cache and Prefix Sharing

Based on the nature of Attention calculation [34], the processing of each token (in both prefill and decoding phase) will generate the corresponding KV context. The computation of latter tokens consume the KV context of earlier tokens to perform Attention calculation. During the processing of a request (i.e., a prompt), the KV context is usually cached in memory for each of the processed token (called KV cache) so that the latter tokens can reuse the history KV context in memory for Attention calculation, rather than recomputing the history KV.

The KV cache can consume a large amount of GPU memory, which can limit the batch size of LLM inference and thus hurt the GPU hardware utilization of the linear layers consist of matrix multiplication (MatMul) calculation. The KV memory size of a single token is $2 \times n_{bytes} \times n_{layers} \times n_{kvhead} \times d_{head}$ Bytes. Take Mistral 7B model [15] with grouped-query attention [3] as an example, which is a small size LLM, each token takes 0.125MiB memory if it uses float16 datatype for KV, and a sequence of 2K length will take 256MiB memory for the KV cache. An NVIDIA A10 GPU with

[3]In this paper, we mainly focus on the basic decoding scheme and will not discuss the parallel decoding technique [4].

24GiB memory can only hold less than 100 such sequences' KV memory, even without considering the memory consumption of the weight itself. The KV memory is usually managed in blocked memory to reduce memory fragment and enable KV sharing [20], the logical continuous KV tensors are mapped to the physical discrete memory blocks mapped with the block table.

When different prompts share the same prefix, the common KV context (K_{prefix}, V_{prefix}) can be computed once and reused by these prompts. The Attention computation between Q and K/V can be performed through the partial Attention on the prefix and the distinct prompt respectively, followed by the Online Softmax [24] to reduce the partial results:

$$\begin{aligned} O_{prefix} &= partial\_attention(Q, K_{prefix}, V_{prefix}) \\ O_{distinct} &= partial\_attention(Q, K_{distinct}, V_{distinct}) \\ O &= online\_softmax(O_{prefix}, O_{distinct}) \end{aligned} \tag{1}$$

The existing LLM inference engines make use of compact prefix tree (i.e., Radix tree) and blocked KV memory to enable KV sharing of the common prefix between different prompts [20, 44]. Due to that different prompts can partially share the same KV context of the prefix, the partial Attention calculation on the prefix (K_{prefix}, V_{prefix}) can be transformed from the matrix-vector mulplication into MatMul to increase the compute intensity [18, 40, 41, 45]. The existing works perform the calculations in Eq.1 in three separated GPU kernels. If there are multiple levels of prefix, each of the extra prefix will introduce two more kernels for the partial Attention and reduction respectively [41].

Besides the saved computation of the common prefix, by storing the KV context of the common prefix with a single copy, it can help to decrease the memory pressure caused by KV memory and potentially increase the allowed batch size.

2.3 Per-iteration Token Batching

Due to the dynamic output length of LLM execution, batching the tasks at the granularity of request results in a waste of the GPU resource, as some requests may finish earlier. Instead, the current LLM inference engines batche the tasks at the granularity of tokens. Specifically, it forms the token-batch at each iteration and allows to add new tokens into the batch dynamically [42], which is also called continuous batching. During the execution, the linear operators of different tokens will be concatenated together into a larger linear operator for execution at each iteration.

Recent works [1, 13] split the long prefill into chunks and mix the prefill chunks with decoding tokens into the same token-batch to conquer the memory-wall problem of pure decoding batches, called SplitFuse or chunked-prefill. By inserting the prefill chunks into the decoding batches, the token number of the batch is enlarged effectively without introducing a lot of memory consumption. As a result, the compute-intensity of the token-batches are increased and the hardware utilization becomes better. Mixing the decoding tokens with prefill chunks is an effective way to increase the throughput, but can potentially increase the latency of the decoding tasks, thus preventing existing LLM inference engines from enabling chunked-prefill by default.

3 OPPORTUNITIES AND INSIGHT

3.1 Emerging Scenarios: Large Batched Through Oriented Tasks with Shared Prefix

LLM has been increasingly used in a broad range of information processing and management tasks [5, 7, 19, 23, 35, 38, 43]. Due to the large traffic of these tasks and the high cost of LLM inference, many of these tasks are processed in very large batch (or even offline) whose performance metric is mainly the throughput rather than latency. Take the snippet generation task [35, 38] in search engine as an example, which is to generate the snippet of the web document (web content) in the search result page according to both document and user query. An example prompt in a previous work [35] is: *generate a short snippet based on the given Document to answer the given Query. Document: <...> Query: <...>*. Due to that it is hard to meet the SLO when performing the massive LLM inferences online, a common practice is to run the snippet generation task offline for the high-frequency web documents and queries, and retrieve the offline results during online serving when the corresponding document-query pair occurs.

Many of the LLM-based information processing and management tasks show the characteristic of the shared prefix between the prompts of different requests, which comes from the nature that different tasks can perform on the same content (e.g., web document). Take the search engine task as an example, it may extract different information from the same web document so that the document can be a shared prefix in the prompt. Specifically, as for the snippet generation task, given that it is to extract the snippet of the document-query pair, the document can be the common prefix if it appears for different queries.

The shared prefix can be managed in multiple levels. For example, the first level prefix can be the global system information and instructions, and the second level can be the web document. However, with the development of LLM and the using of task-specific model fine-tuning, the prefix of system information and instructions becomes shorter and shorter, and the length of the prefix will finally be dominated by the web document itself. On the one hand, the LLM models are incorporating more and more reasoning abilities into themselves (like OpenAI o1 [25]) and will not rely on complex prompts. On the other hand, the task-specific fine-tuning can help to incorporate the complex prompts into the model weight. As a result, the prefix sharing of the long context rather than the other instructions can be the most important for many of the information processing and management tasks. This motivates the first-level prefix enlargement in Sec.4.2.

3.2 The New Optimization Demands

As discussed in Sec.2, the existing works have proposed solutions to support prefix sharing and per-iteration token batching to support LLM inference. However, the state-of-the-art solutions still lack specialized optimization for the increasingly import large batched scenarios described in Sec.3.1, showing three main limitations:

1. The implicit prefix caching cannot lead to the optimal KV reuse for the large batched inference. The existing LLM serving systems [20, 44] use the prefix tree to maintain the KV blocks. The same prefix (identified by runtime hashing) can be

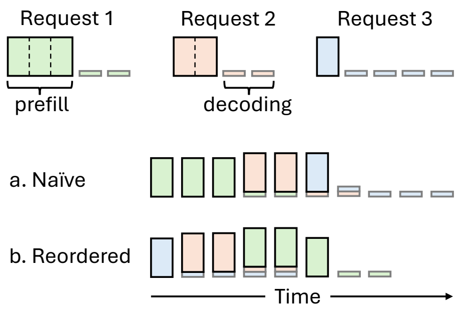


Figure 1: The effect of processing order of requests with chunked-prefill enabled. Given the three requests with different prefill/decoding length characteristics, the naive token batching in the coming order of the requests has worse token mixing of decoding and prefill chunks.

mapped to the same KV blocks to avoid repeated computation. It uses the LRU policy to evict the blocks from the block table. As for a large batch of inputs to be processed, it does not keep the prefix in memory in the global view in the large batch, but only according to the history input. As a result, the shareable KV blocks can be easily evicted prematurely. We have evaluated vLLM's prefix sharing with a typical industry workload. The optimal token saving ratio by prefix reusing is 56% by analyzing the input dataset. Note that the saving ratio is calculated by:

$$R_{saving} = (1 - \frac{N_{processed\_prefill\_tokens}}{N_{logical\_prefill\_tokens}}) \times 100 \quad (2)$$

$N_{processed\_prefill\_tokens}$ is the number of prefill tokens processed after prefix sharing, and $N_{logical\_prefill\_tokens}$ is the original prefill token number of all the requests. The vLLM's implicit prefix caching only achieves 44.2% (refer to Sec.6.3.1).

There is a demand to enable better prefix sharing for the large batched tasks, especially considering that all the prefill characteristics in the batch are known ahead.

2. The online serving oriented scheduling and token batching can be suboptimal for the prefix-shared and throughput oriented tasks. The current LLM inference systems schedule the tasks at the granularity of request. They do not analyze the prefix sharing characteristics of the whole large batch nor schedule the requests sharing common prefix together. This not only can evict the shareable KV context prematurely as discussed above, but also extends the lifetime of the shareable KV memory, exacerbating the problem of large KV memory consumption. Besides, these systems design tends to support online services. They form the token-batches at each iteration with the constraint of requests' arriving order, which can result in suboptimal token-batches. Take the example in Fig.1, the naive scheduling in the order of request's coming cannot mix the decoding of *request 3* with the prefill chunks of other requests (chunked-prefill background in Sec.2.3). Instead, by scheduling *request 3* earlier, its decoding can be mixed with the prefill chunks of other requests. Another problem is that, the current systems use the token and request number in the token-batch as the threshold to limit the batching, which limits to batch more tokens together in the decoding dominated token-batches

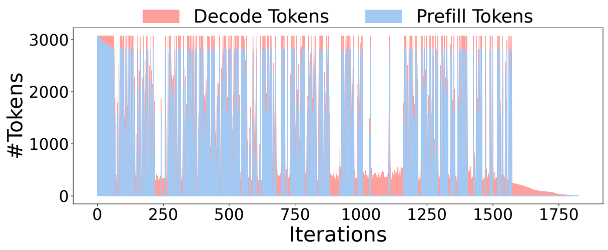


Figure 2: The token number in the batch processed at each iteration for an industry task with vLLM's chunked-prefill. It has "valleys" for many iterations.

and prevents better utilizing the GPU even when the memory is enough. Fig.2 shows the number of tokens at each iteration for a typical industry task (Sec.6.2.3) performing with vLLM's best configuration. It shows that there are "valleys"[4] at many iterations where the number of tokens may not be large enough to saturate the GPU.

There is a demand to reschedule the tasks to make the requests sharing common prefix closer, and enable better mix of decoding tokens with prefill chunks. There is also a room to enlarge the size of token-batches dominated by the decoding tokens.

3. There is still room for Attention optimization of prefix sharing. As described in Sec.2.2, the existing works [18, 40, 41, 45] support prefix-shared Attention by computing the Attention on different chunks of KV, performing MatMul on the common prefix and the basic matrix-vector multiplication on the non-shared chunk. But the computation on different chunks are in different kernels. On the one hand, the separated kernels increase the tail effect of GPU kernel. On the other hand, the multiple kernels

3.3 Insight

To optimize the throughput oriented tasks described in Sec.3.1 and address the limitations described in Sec.3.2, by leveraging the characteristics of the large batch, BatchLLM has the following insights.

1. Explicit prefix identification and sharing. Instead of hashing and matching the common prefix at runtime and manage the prefix with LRU cache, BatchLLM explicitly identifies the common prefix globally within the large batch ahead-of-time, which will not miss the opportunity of prefix sharing caused by the implicit cache. Besides, BatchLLM refactors the prefix tree with a Dynamic Programming algorithm to enlarge the first-level prefix to avoid the system complexity and kernel overhead of the multi-level prefix. Details are described in Sec.4.2.

2. Grouped scheduling, request reordering and memory-centric token batching. BatchLLM schedules the requests at the granularity of a group of requests sharing the common prefix, which makes the prefix sharing convenient and shrinks the lifetime of the prefix's KV memory. BatchLLM reorders the requests according to the length of the prompt and the estimated decoding length. As indicated in Fig.1, BatchLLM will schedule the requests with larger ratio of decoding length to prompt length with higher priority. In

[4]This paper uses *valley* to express the token-batches that has smaller number of tokens, appearing as the valley in the timeline of token numbers.

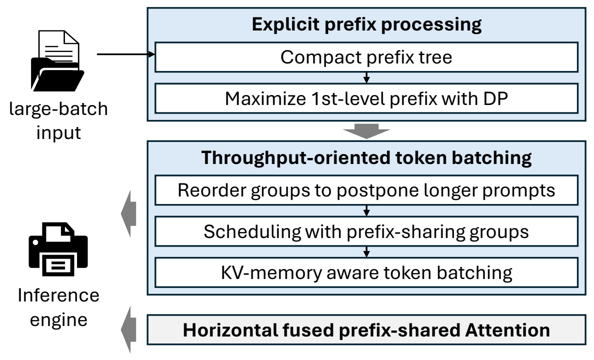

large-batch input
Explicit prefix processing
Compact prefix tree
Maximize 1st-level prefix with DP
Throughput-oriented token batching
Reorder groups to postpone longer prompts
Scheduling with prefix-sharing groups
KV-memory aware token batching
Inference engine
Horizontal fused prefix-shared Attention

Figure 3: BatchLLM overview.

this way, the longer prompts will be scheduled later and can be better mixed with the earlier decoding tokens. It also forms the token-batches with the consideration of the KV memory usage, allowing to batch more tokens together to increase the size of token-batches and reduce the "valleys" in Fig.2.

3. Attention kernel optimization with horizontal fusion. BatchLLM horizontally fuses the computation on different KV chunks into the same kernel to reduce the tail effect and kernel launch overhead. This method is not novel, but effective.

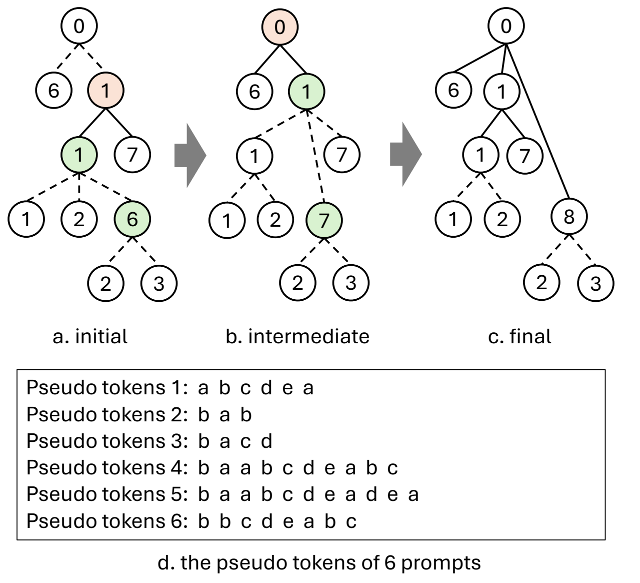
4 DESIGN METHODOLOGY

4.1 Overview

Fig.3 shows the overview of BatchLLM optimizations. First, it analyzes the large batch of input prompts and identifies the common prefix explicitly. This is done before the scheduling of the requests. It simplifies the multi-level prefix into a single level prefix with the insight that the prefixes of the information processing and management are usually dominated by a long context (e.g., web document), as discussed in Sec.3.1. The explicit prefix processing is illustrated in Sec.4.2. The requests will be organized into groups where each group corresponds to the quests sharing the same prefix. The group will be the basic unit of task scheduling. Before scheduling the groups, BatchLLM will also reorder the groups according to the ratio of prefill length and estimated decoding length and postpone the groups with longer prefill and shorter decoding. It then forms the token-batches with the consideration of the KV memory usage. This aims to better mix the decoding steps with the prefill chunks to increase the overall token-batch size. The throughput-oriented scheduling and token-batching optimization is described in Sec.4.3. BatchLLM also implements the horizontal fused Attention kernel to reduce the tail effect and kernel launch overhead, described in Sec.4.4. We have integrated the above optimization into the state-of-the-art LLM inference engine (vLLM [20]).

4.2 Explicit Global KV Reuse Identification

As discussed in Sec.3.1, different prompts may have multiple levels of prefixes. The multi-level prefixes can lead to system challenges and overhead of token batching and fused Attention kernel. For the token batching, it requires to manage the dependencies of the different levels of prefixes. For the Attention kernel, it will introduce



Pseudo tokens 1: a b c d e a
Pseudo tokens 2: b a b
Pseudo tokens 3: b a c d
Pseudo tokens 4: b a a b c d e a b c
Pseudo tokens 5: b a a b c d e a d e a
Pseudo tokens 6: b b c d e a b c

Figure 4: The preprocessing to maximize the first level prefix reusing. It converts from the initial prefix tree (a) to the final prefix tree (c). Each cycle represents a node containing a number of tokens. The number in the cycle is the token number of the node. It iterates bottom-up to maximize the first level reusing recursively until reaching the root node, which is a dummy node.

more GPU kernels of the multi-level reuse. Instead, BatchLLM compresses the multi-level prefixes into a single level prefix, to avoid the above challenges. This is based on the insight that the length of the prefixes are usually dominated by a single level due to the stronger reasoning ability of the newer LLMs and the task-specific fine-tuning (e.g., the prefix corresponding to the web document body can be much longer than other levels of the prefix, as discussed in Sec.3.1).

As discussed in Sec.3.3, BatchLLM explicitly identifies the common prefix among the prompts within the large batch. It makes use of the compact prefix tree[5] to identify and represent the common prefix between prompts. However, directly using the first level prefix of the basic prefix tree as the common prefix can lead to suboptimal prefix reusing. Take the pesudo prompts in Fig.4d as an example. The 6 prompts are organized with the compact prefix tree in Fig.4a, where each node represents several consecutive tokens of the prompt (the number in the node means the number of tokens) and the tokens in a node is shared by its children. The right child of the root node (i.e., token 'b') is shared by 5 prompts in total (prompt 2 to 6) according to Fig.4a, thus reusing the first level prefix can save 4 tokens' KV computation. However, it is easy to infer that sharing the first 8 tokens between prompt 4 and 5 can save 8 tokens' KV computation. Thus it requires to refactor the naive prefix tree to maximize the achieved first-level token reusing.

[5]The compact prefix tree, also called Radix tree, is the prefix tree in which each node that is the only child is merged with its parent.

Algorithm 1 Dynamic Programming on the prefix tree to maximize the first-level token reusing of node D.

```
1: ▷ Note the 1st-level prefixes of a node are its children's tokens
2: function MaximizeReuse(D)
3:     if D has no children then
4:         return
5:     for $child \in D.children$ do
6:         MaximizeReuse($child$)                ▷ Solve sub-problems
7:         for $gchild \in child.children$ do
8:             $gain \longleftarrow$ (leaves($gchild$) - 1) $\times$ tokens($gchild$)
9:             $penalty \longleftarrow$ tokens($child$)
10:            if $gain > penalty$ then
11:                Fork $child$ and merge with $gchild$
```

We design a Dynamic Programming algorithm on the compact prefix tree to maximize first-level token reusing, shown in Algo.1. The basic insight is that, given a node D to maximize its first-level token reusing, it first solves the sub-problems of all D's children to maximize each child's first-level token reusing (line 6 in Algo.1), it then try to merge D's first-level prefix with its children's to see if it can enlarge D's token reusing (line 7-11 in Algo.1). It recursive performs the above procedure until reaching the root note, by calling *MaximizeReuse*(*root*). Fig.4b and Fig.4c shows an example of this procedure. In Fig.4b, the left node in level 3 of the tree is forked and merged with it's right child, which enlarges the 1st-level prefix reusing of the right node in level 2. Similarly, the right node in level 2 is forked and merged with its second child to enlarge the 1st-level prefix reusing of the root node. After maximizing the first-level prefix, the other levels of prefixes will be expanded during the token batching and will not be regarded as the shared context.

We have compared the token saving ratio between the original multi-level prefix and the enlarged single prefix. The saving ratio is calculated according to Eq.2. For a dataset of the snippet generation task, the saving ratio of the original multi-level prefixes is about 56%, and the ratio is about 55% for the first-level prefix after enlarging (Sec.6.2.2). It means the enlarged first-level prefix only has 1% loss of the token reuse compared to the multi-level prefix for this typical workload, with the advantage of reducing the complexity and the overhead of the token batching and Attention kernel optimization.

4.3 Throughput-oriented Token Batching

4.3.1 Prefix-sharing Group Based Scheduling. As described in Sec.4.2, BatchLLM identifies the common prefix explicitly ahead-of-time. To simplify the KV reuse and reduce the lifetime of the common prefix in the memory, BatchLLM schedules the requests sharing the same prefix all together. Specifically, the procedure described in Sec.4.2 will generate the *prefix-sharing groups* and BatchLLM will schedule the requests at the granularity of *prefix-sharing group* rather than each single request. A *prefix-sharing group* corresponds to the collection of requests sharing the same prefix. In this way, the requests sharing the same prefix can start at the same time and the memory of the common prefix can be released as soon as the group is completed. This differs from the LRU cache and reference counter based prefix management [20, 44] and can make sure all the common prefix can be reused the best and the memory will not be retained unnecessarily.

To realize the *prefix-sharing group* based scheduling, BatchLLM maintains three queues of the to-be-batched tokens: common prefix queue, distinct prompt queue, and decoding queue. At each iteration, BatchLLM will form the token-batch by fetching the tokens from the three queues in the order of decoding queue, distinct prompt queue, and common prefix queue (one constraint is that the a common prefix should be fetched and processed before its corresponding distinct prompts). On the one hand, fetching the decoding tokens before prefills can help to better mix the two types of tokens together in the token-batch [13]. On the other hand, fetching the decoding and distinct prompt tokens before the prefix tokens can make sure the active requests (i.e., the request that has been partially processed, either partial prefill chunks has been processed or already in decoding phase) can be scheduled before the inactive requests, which helps to finish the active requests earlier and thus release their memory earlier.

4.3.2 Request Groups Reordering. As discussed in Sec.3.3, BatchLLM reorders the input requests to schedule the requests according to the ratio of prompt length to decoding length (R). We define the ratio R as:

$$R = \frac{L_{decode}}{L_{prefill}} \tag{3}$$

In Eq.3, $L_{prefill}$ is the number of prefill tokens and L_{decode} is the length of the output tokens. The requests with larger R will be scheduled earlier. This helps to issue the requests with long decoding steps earlier and make the latter prompts's length larger and thus can be better mixed with the previous decoding tokens to enlarge the token-batch size (example in Fig.1). The challenge is that the exact number of output tokens is not known before finishing the execution. We profiled the dataset of several typical information processing tasks (snippet generation, offline ranking, ads, etc.) and observed that the distribution of output length is relatively concentrated for a task. We thus assume the output length is constant and normalize $L_{prefill}$ into 1 in Eq.3 for all requests uniformly. As we schedule the requests in the unit of *prefix-sharing group*, the ratio of the group (after normalizing the length of output) is

$$R_{group} = \frac{1}{L_{prefix} + \sum L_{distinct}} \tag{4}$$

According to Eq.4, the request groups with larger R_{group} will be scheduled with higher priority.

4.3.3 Memory-centric Token Batching to Saturate GPU. The request group reordering described above helps to reduce the "valleys" in the timeline of the token numbers ("valley" example in Fig.2) by providing more chances for the decoding tokens to be able to find the prefill chunks to mix together. Another limiter of the size of token-batches is that, the existing works [20] use a fixed request number to limit the token batching, i.e., the request number cannot exceed a fixed threshold within a token-batch. For a token-batch dominated by the decoding tokens, it is easy to reach the upper bound of the request number while still have not achieved a large number of the total tokens. For example, vLLM sets the threshold of request number as 256 by default, if a token-batch already has 256 decoding tokens, it will have no chance to add more prefill chunks into this token-batch even if there is still enough memory to hold the KV memory of the prefill chunks. Note that purely increasing this threshold does not solve the problem in practice, as noted by the vLLM community[6]. This is because directly excessively enlarging batch size may cause the engine to have more frequent memory swaps and recomputation due to the GPU memory can be not enough to hold the new coming KV tensor.

The target of the token-batching is to enlarge the number of tokens in the batch under the constraint of GPU memory size. Thus there are two main factors that matters for the token-batching procedure: whether the number of tokens is large enough in the batch to saturate the GPU, indicating the current status of the token-batch; and whether the remaining memory is enough to accommodate more prefill chunks, indicating if the status can be improved. With this insight, BatchLLM uses the KV memory itself and the predetermined chunk size S_{chunk} as the limiter rather than using the indirect limiter of the request number.

Specifically, BatchLLM predefines a size as the upperbound of the KV memory size ($M_{threshold}$). When deciding the token-batching at each iteration, it will only check whether adding a prefill chunk will exceed $M_{threshold}$ or not, without considering the number of requests. The prefill chunk will be added into the token-batch if the remaining GPU memory capacity allows it. Algo.2 describes the detailed procedure. Note that we also use a fixed number to limit the per-batch token number for token-batching decision (noted as the chunk size S_{chunk}), which aligns with that in vLLM and helps to distribute the token numbers in different iterations more even.

Algorithm 2 Simplified memory-centric token batching.

```
Require: Chunk size $S_{chunk}$, KV memory threshold $M_{threshold}$.
1: $RaggedBatch$ is the current token-batch containing the active
   decoding tokens
2: $M_{current}$ is the current usage of KV memory
3: while $|RaggedBatch| < S_{chunk}$ and $M_{current} < M_{threshold}$ do
4:     if prefill request $p$ available then
5:         Add $p$ to $RaggedBatch$
6:         $M_{current} \leftarrow M_{current} + M_p$
7:     else
8:         Break
9: return $RaggedBatch$
```

4.4 Prefix-shared Attention Optimization

As described in Sec.3.2, the existing works use separate kernels to compute the partial Attention on the prefix and the distinct KV context. This will lead to the tail effect of each kernel and the launch overhead of these kernels. BatchLLM horizontally fuses the two partial Attention calculation into one kernel. Different parts of the computation are performed in different thread blocks. Note that the problem shape of the two parts are different. BatchLLM applies different tiling configuration for the two parts to achieve the best performance. It uses the common auto-tuning method to find the best configuration.

The horizontal fusion itself is not novel, but can effectively help to increase the performance of the prefix-shared Attention.

[6]https://github.com/vllm-project/vllm/issues/6801

5 IMPLEMENTATION

We implement the prefix identification (Sec.4.2) in Python as a standalone preprocessing script. It accepts a list of inputs and generate a list of *prefix-sharing groups*. The inputs can be the nested input ids (e.g., `[<list of input 0 tokens>, <list of input 1 tokens>, ...]`), and each *prefix-sharing group* is represented as a tuple of the prefix and the corresponding distinct prompts (e.g., `tuple(<list of prefix tokens>, [<list of distinct 0 tokens>, <list of distinct 1 tokens>, ...])`).

We integrate the token batching optimization (Sec.4.3) in vLLM [20] v0.5.4. Specifically, we customize the vLLM to accept the list of *prefix-sharing group* tuples generated by the preprocessing script, and implement the group-wised scheduling and token batching logic upon the vLLM token batching function.

Finally, we integrate the prefix-shared Attention optimization (Sec.4.4) as the Attention kernel backend of the customized vLLM. We implement the Attention in BatchLLM through OpenAI Triton language [33]. The limitation of the Triton based implementation is that, the performance of Triton version of FlashAttention is still worse than that of CUDA version (Sec.6.3.3). However, the advantage is that Triton supports multiple platforms (both NVIDIA and AMD GPUs) so that BatchLLM implementation can run on both NVIDIA and AMD GPUs. We use the builtin autotuner of Triton to tune the tiling size of the Attention implementation in BatchLLM. Besides that, some ahead-of-time compilation methods are applied to reduce the launching overhead, including warm-up process on NVIDIA GPUs, and AOTriton[7] on AMD GPUs.

6 EVALUATION

6.1 Setup

Baseline Specification. We compare BatchLLM with vLLM, the state-of-the-art LLM inference engine with prefix sharing, for the end-to-end comparison to demonstrate the efficacy (Sec.6.2). We integrate all our design in vLLM v0.5.4 and the baseline for comparison is also vLLM v0.5.4 (except as noted). We use FlashAttentionn [6] (based on v2.6.1) as the Attention backend[8] of the vLLM baseline in NVIDIA GPU environments.

We enable the *prefix-caching* and *chunked-prefill* in vLLM baseline for the end-to-end comparison in Sec.6.2. The *prefix-caching* uses the implicit LRU-based caching policy. We use 2048 as the token-batch size when enabling the chunked-prefill of vLLM baseline, which is a recommended size to achieve its better throughput according to the vLLM community. Note that the prefix-caching and chunked-prefill are still not supported to be enabled together in v0.5.4. To make a fair comparison, we optimized the code of vLLM 0.5.4 baseline to make it able to enable the two techniques together, which makes the vLLM baseline have better performance in Sec.6.2.

For the kernel comparison in Sec.6.3.3, we compare BatchLLM with Cascade-Inference [41], which is the SOTA implementation of the prefix-shared Attention on NVIDIA platform. It is worth

[7]https://github.com/ROCm/aotriton

[8]We do not use Flashinfer [41] as the backend as we observe it will result in incorrect results of the end-to-end output in vLLM v0.5.4. Note that the single kernel result of Flashinfer is correct, so that there may be some engineer problem of its integration to vLLM v0.5.4.

noting that Cascade-Inference is similar to our design on kernel-levels. One difference between Cascade-Inference and BatchLLM is that BatchLLM applies horizontal fused kernel as described in 4.4. Besides, Cascade-Inference cannot be applied on the non-NVIDIA backend as it is developed with CUDA, while BatchLLM is based on OpenAI Triton and can support both NVIDIA and AMD GPUs.

All the other packages like PyTorch and Huggingface transformers are dependent to the docker file of vLLM v0.5.4. The OpenAI Triton version is 3.0.0.

Hardware and System Specification. We conduct our experiments on both NVIDIA A100 80GB GPU and AMD MI200 GPU. The CUDA version on the NVIDIA platform is 12.1. The ROCm version on the AMD platform is 6.1. The CPU is AMD EPYC 7V12 CPU @2.45GHz. The operation system is Ubuntu 20.04.

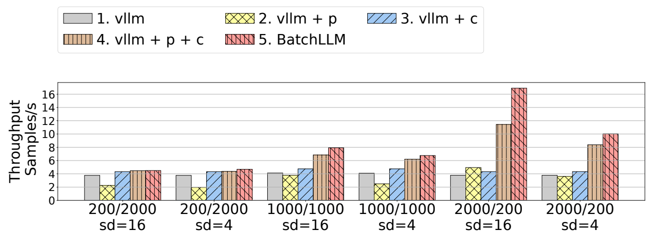
Models. We evaluate BatchLLM and the baselines with Qwen2-7B [39], Mistral-7B [15] and Llama3-8B [2] models. These are among the most popular models with the size affordable for the information processing and management systems dealing with massive inputs.

Dataset. We evaluate BatchLLM and the baselines with both three microbenchmarks with different data distributions and two typical industry tasks.

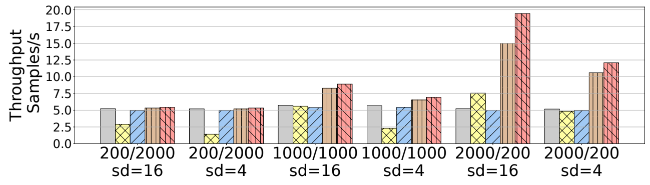
As for the microbenchmarks evaluation, we use three different settings of the length of common prefix and distinct prompt. The length of prefix/distinct are 2000/200, 1000/1000 and 200/2000 respectively. Note that the lengths are typical for the industry workloads. We evaluate the three microbenchmarks under different share-degrees. The share-degree means the number of distinct prompts sharing the same prefix. Specifically, we conduct the evaluation of the three microbenchmarks with sharing degree 4 and 16, respectively. The microbenchmark evaluation is in Sec.6.2.1.

As for the industry tasks, we conduct the evaluation on two typical scenarios in web search engines. The one is that, given a document (as the shared prefix), the task is to extract the corresponding information according to the user query (e.g., snippet generation task). The other is that, given a user query (as the shared prefix), the task is to rank a list of documents according to some metrics (e.g., offline ranking). These two scenarios have different data distribution so that it can better demonstrate the efficacy of BatchLLM. Specifically, the common prefix of the former task is much longer than the distinct prompt, and the latter task is exactly the opposite. We construct two datasets synthesized according to two of the above scenarios in industry, and make the case study in 6.2.2 and 6.2.3.

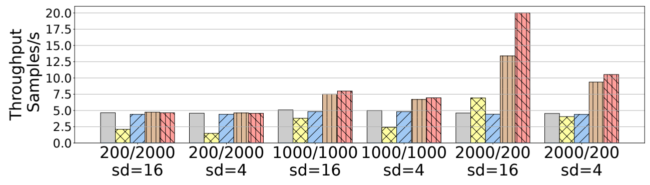
We conduct the breakdown experiments in Sec.6.3 to demonstrate the efficacy of each techniques in this paper.



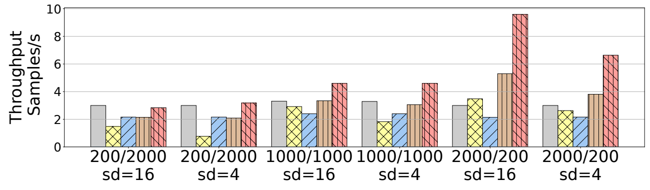
(a) Llama3-8B Throughput, on A100.



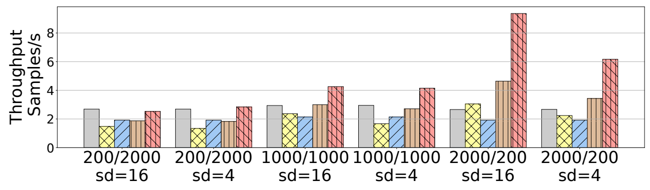
(b) Qwen2-7B Throughput, on A100.



(c) Mistral-7B Throughput, on A100.



(d) Qwen2-7B Throughput, on MI200.



(e) Mistral-7B Throughput, on MI200.

Figure 5: Microbenchmark evaluation of different models and hardware enviromments. The setting *m/n* (like *200/2000*) indicates the length of shared prefix and non-shared context. *sd* means *shared-degree*, showing that how many requests in one *prefix-sharing* group with the same shared prefix. In the legend, the setting with '+ p' means *prefix-caching* enabled, while '+ c' means *chunked-prefill* enabled.

6.2 End to End Comparison

6.2.1 Microbenchmark Evaluation. Fig.5 shows the end-to-end throughput comparison between BatchLLM and the baselines. In these benchmark experiments, we compare BatchLLM with vLLM of different configurations, under the FlashAttention backend [6], and with or without *prefix-caching* and *chunked-prefill* enabled.

The results show that BatchLLM outperforms the vLLM baseline under different configurations, for different models on different GPU platforms with different input data distributions. Specifically, when comparing with *vLLM + prefix-caching + chunked-prefill*,

Table 1: Industry workload evaluation.

Settings	Case 1 (CUDA)	Case 2 (ROCm)
vLLM	7.13	4.34
+ prefix-caching	9.9	4.41
+ chunked-prefill	7.08	4.56
+ prefix-caching + chunked-prefill	9.83	5.88
BatchLLM	12.85	8.65

which performs the best across all vLLM's configurations, BatchLLM can get 1.1-1.5× speedup on A100 and up to 2× speedup on MI200. It also shows that the speedup becomes more significant when the portion of the common prefix becomes larger. This is because the reuse ratio becomes larger and BatchLLM can save more computation and memory for this case. The 200/2000 dataset has little room of KV reusing and thus all the optimizers do not show significant performance improvement.

The performance improvement of BatchLLM comes from the better KV reuse with the explicit prefix identification, the token-batching to better mix the decoding with prefill chunks together, and the better Attention kernels, which we will analyze through the breakdown in Sec.6.3.

6.2.2 Case Study 1: Long Common Prefix with Short Distinct Prompt. This industry workload is similar to the snippet generation task described in the recent work [35], which is also described in Sec.3.1. The original input has two levels of prefixes: the global shared instruction to extract the information from the document-query pair, and the shared document between different groups of queries. The first-level prefix is very short. With the optimization described in Sec.4.2, the first-level prefixes are enlarged (i.e., some of the second-level prefix of the document is merged into the first-level prefix). We have analyzed the token saving ratio according to Eq.2. It shows that the saving ratio are nearly the same between the multi-level and the enlarged single level prefixes, 56% for the former and 55% for the latter.

We synthesize the dataset for the evaluation according to the data distribution of the industry workload, where the average share-degree (i.e., the number of the distinct prompt that share the same prefix) is about 7, the average prefix length is about 1100 tokens, and the average distinct prompt length is about 400 tokens. We sample 3000 queries for this case study.

We conduct this experiment on the NVIDIA A100 GPU, using Mistral-7B model. Table.1 shows the effectiveness of BatchLLM of this workload (and another industry workload that will describe later), with about 1.3× speedup over the best configuration of vLLM. BatchLLM better reuses the KV context than the vLLM baseline with the optimization in Sec.4.2, for which we have the breakdown analysis in Sec.6.3.1. BatchLLM also better mixes the decoding tokens with prefill tokens to increase the size of token-batches with the optimization in Sec.4.3, for which we have the breakdown analyze in Sec.6.3.2. It also benefits from the optimized Attention kernel described in this paper.

6.2.3 Case Study 2: Short Prefix with Long Distinct Prompt. We have evaluated another industry workload with different data distribution to that in Sec.6.2.2. This is a ranking job that ranks a set of web documents given the user query according to the relevance. Specifically, if forms the query-document pairs where the same query along with the instruction is the shared prefix across a number of documents. The difference to the workload in Sec.6.2.2 is that, the length of the common prefix is shorter than that of the distinct prompt (i.e., document). However, the sharing degree in this case is relatively large. We evaluate this workload with 1000 samples, where the average share-degree is 11.36, the length of the common prefix is about 770, and the length of the distinct prompt is about 978.

We conduct this experiment on the AMD MI200 GPU, using Qwen2-7B model. BatchLLM still gets a significant gain comparing with all the other vLLM settings as shown in Table.1, 1.47× speedup than the best performance of vLLM. Even though the shared prefix has a shorter length, the large sharing degree still makes it achieve a high token saving ratio, which is about 38%. As a result, it still benefits from the explicit common prefix reusing.

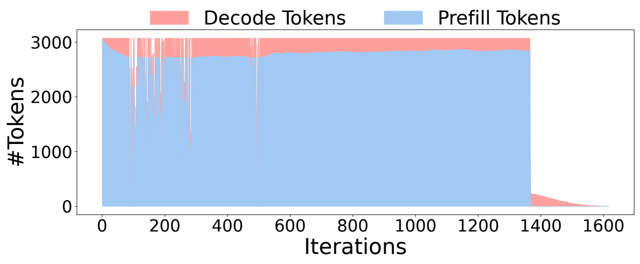


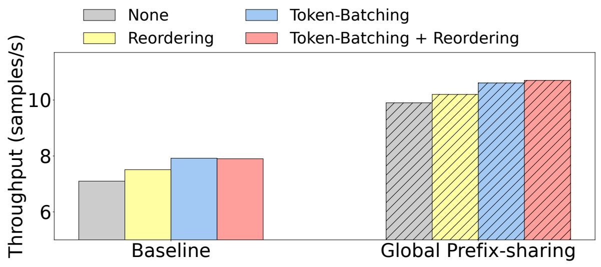
Figure 6: The token number at each iteration with the token-batching optimization in Sec.4.3, for the same workload with that in Fig.2. It significant reduces the "vallyes" compared to the baseline in Fig.2. Note that the prefix-sharing is not enable here, showing the efficacy to the general scenarios of batched/offline inference.

6.3 Breakdown Analysis

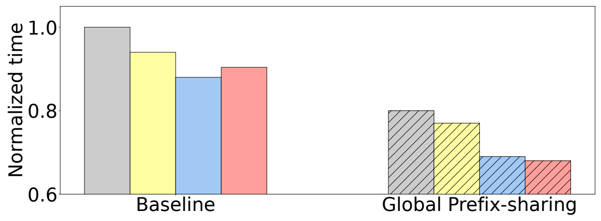
6.3.1 Reusing Effect Comparison. We compare the saving ratio (refer to Eq.2) between vLLM's prefix-caching and BatchLLM on the industry task in Sec.6.2.2. The saving ratio of vLLM's prefix-caching is 44.2%, and BatchLLM achieves 54.9% for this metric, showing that the explicit prefix identification can help to reuse more KV context. Note that we only use 3000 requests to evaluate the saving ratio. With the increase of the requests number, it can be expected that the implicit LRU cache policy can suffer more from the eviction of the reusable KV context.

6.3.2 Token-batching Analysis. Fig.6 shows the per-iteration token number, using the same workload with Fig.2. It clearly shows that the token-batching optimization in BatchLLM successfully reduces the "valleys" of the iterations, thus can increase the overall GPU utilization.

To investigate the transferability of token-batching optimization in Sec.4.3, we conducted ablation experiments on the memory-centric token-batching and group-based reordering/scheduling, as shown in Fig.7. Meanwhile, we propose a metric suitable for Large Batched LLM data processing, namely the *end-to-end serving time*

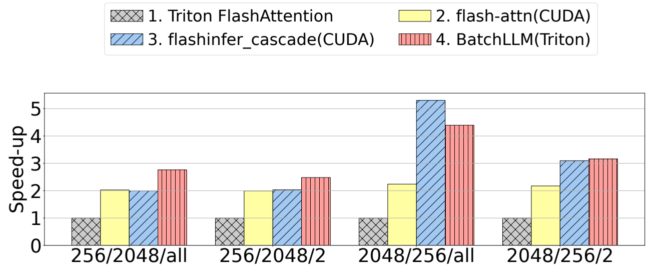


(a) The throughput comparison.

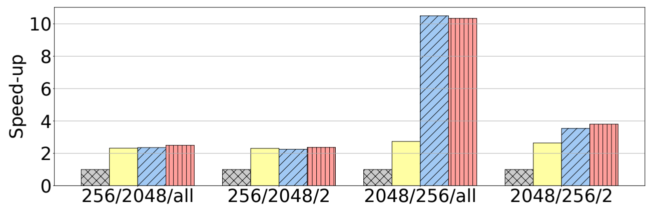


(b) The normalized time comparison excluding the cool-down (*TECD*).

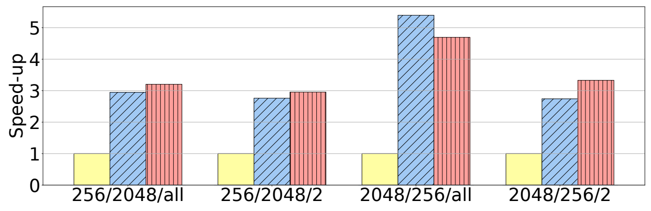
Figure 7: The performance comparison between enabling token-batching or reordering and the baseline. The bar of "None" is the vLLM 0.5.4 baseline with or without BatchLLM's global prefix-sharing optimization. The left side is the vanilla vLLM and the right side is with BatchLLM's prefix-sharing.



(a) Kernel performance under *decoding* scenarios, #request = 32.



(b) Kernel performance under *decoding* scenarios, #request = 256.



(c) Kernel performance under *chunked-prefill* scenarios, with #prefill-request = 7, and #decoding-request = 256.

Figure 8: The performance comparison between the baseline, CascadeInfer and BatchLLM. For one chunked-prefill batch or one decoding batch, the setting *i/j/k* (like *256/2048/all*) means the length of *global shared prefix*, the length of *distinct parts* for each request, and *how many requests* with one single shared prefix in one single *prefix-sharing group*.

excluding the cool-down (TECD). This metric excludes the cool-down phase of the service (as shown in the rightmost part of Fig.6), because as the data scale increases, the time of the cool-down phase remains constant (it is related to the number of iterations of the few remaining requests in the engine). By profiling *TECD*, it can better reflect the actual processing time of ultra-large-scale data with the same length distribution from a batch of smaller-scale data.

Fig.7a shows the throughput improvement of enabling group reordering (Sec.4.3.2), memory-centric token batching (Sec.4.3.3), and the both. We conducted experiments using Mistral-7B on the task described in Sec.6.2.2 (with 3000 requests), and achieved a throughput improvement of up to 9%. The left side is the baseline that does not enabling the prefix-sharing. This is to demonstrate that this optimization is general and can be applied to the non-shared workloads. The right side is for the prefix-sharing scenario. It shows that each of the two techniques can achieve better performance than the valina strategy (the "None" bar in the figure). And the two optimization can work together well for the prefix sharing scenarios.

According to Fig.7b, by applying strategies of BatchLLM token-batching, we can reduce the *TECD* time by up to 15%, and to some extent compress the overall number of steps excluding the cool-down (about 1600 -> about 1350) while keeping the batch size almost unchanged.

6.3.3 *Kernel Performance Comparison.*

We compare BatchLLM's prefix-shared Attention implementation with several baselines on both NVIDIA A100 GPU and AMD MI200 GPU. Fig.8 shows the performance comparison of different implementations, including Triton official FlashAttention implementation[9], official FlashAttention[10], FlashinferCascade [41], and BatchLLM's kernels. We compare the performance among different kernels under two scenarios: the pure decoding scenario (request count 256) and the chunked-prefill scenario where a token-batch includes both prefill chunks and decoding tokens (7 prefill requests and 256 decoding requests). Besides that, how many requests one single *prefix-group* have, and various *length* setting of both shared prefix and non-shared tokens, are non-trivial. Our experiments are conducted based on these insights.

[9]https://github.com/triton-lang/kernels/blob/main/kernels/flash_attention.py
[10]https://github.com/Dao-AILab/flash-attention

According to the results, we can see that the Triton-based FlashAttention still has a performance gap to the CUDA-based FlashAttention, although we have already tuned the hyparameters using Triton's autotuner. (Note we omit the performance under chunked-prefill setting because it's even worse.) This indicates the high-level Triton compiler still have a room to achieve the performance of low-level CUDA compiler for complex computations (for example, we observe more shared memory bank conflict of the Triton FlashAttention than the CUDA version).

However, based on the insufficient performance of Triton, our kernel still shows good performance compared to the Cascade-Inference kernel. BatchLLM's kernel can still achieve similar or better performance comparing with all these CUDA implementations, showing the potential of the kernel optimization in BatchLLM.

Generally, our kernel performs competitively, while some bad cases cannot be ignored. Take the scenaro where there are multiple requests with the same sharing prefix, and the common part is much longer than the distinct part, as an example. For these long-key-value cases, split-k could be a solution through which the calculation units on GPUs could be fully utilized. We'll still focus on how to make optimization in the future.

6.3.4 Global Prefix Identification Overhead. We have measured the time of the global prefix identification described in Sec.4.2. The time range of the overhead starts before adding the token ids of each prompt to form the basic prefix tree and ends after generating the final list of prefix-sharing groups. As for the industry workload in Sec.6.2.2, the total overhead of the 3000 requests is 0.28 seconds. While the time to process the 3000 requests with the input of prefix-sharing groups format is 233.46 seconds. The global prefix identification procedure nearly has no overhead compared with the end-to-end execution time.

7 RELATED WORK

Prefix sharing systems. The vLLM [20] proposes the PagedAttention to manage the KV memory in blocks, which enables to reuse the KV context both intra and inter requests in memory block level. Prompt Cache [12] and SGLang [44] propose the domain specific language (DSL) to define the shared prefix of prompt. Besides, SGLang proposes the radix tree based KV cache management with LRU policy for KV memory eviction. RAGCache [16] studies the common KV caching for RAG optimization. Some recent works [10, 14, 17, 26, 28] study the distributed request scheduling with the consideration of prompt prefix reuse. None of these works identify the common prefix ahead-of-time for the large batch and enable the explicit reusing. Instead, BatchLLM manages the prefix reusing explicitly and can reuse the KV context the best for the large batched scenario.

Token batching and serving optimization. Orca [42] proposes the continuous batching method for transformer models with the per-iteration token batching of auto-regressive models [11]. FastGen [13] and SARATHI [1] study the mix of of prefill chunks and decoding tokens into the same token-batch to increase the compute intensity of the batch. FlexGen [31] proposes the swizzled token computation and offloading to support the LLM inference with limited GPU memory. Some works [9, 30] study the request scheduling to achieve better SLO for online LLM serving. BatchLLM differs from these works as it targets the large batched inference, leveraging the static information to reorder the requests and using the memory-aware scheduling to enlarge the size of token-batch. The vLLM [20] uses the multi-step scheduling method to reduce the token-batch scheduling overhead, which is orthogonal to BatchLLM. Another orthogonal dimension to improve the token-batch performance is the pruning and quantization [8, 22, 36], which can be applied concurrently with the optimizations in this paper.

Attention kernel optimization. Memory-efficient Attention [29] and FlashAttention [6] fuse the Attention operators into a single kernel to boost the performance, with Online Softmax [24] to tile the Attention computation. The recent prefix-shared Attention optimization (ChunkAttention [40], RelayAttention [45], Hydragen [18], and Cascade Inference [41]) compute the attention on the prefix and other part separately in different kernels, for which the former is transformed from matrix-vector multiplication between Q and K/V into MatMul, and reduce the results of different parts according to the Online Softmax algorithm. Different from these works, BatchLLM fuses the Attention computation of different parts into the same kernel to reduce tail latency and kernel launch overhead. The horizontal fusion has been proposed and used in the existing machine learning optimizers [21, 27]. BatchLLM borrows this idea and applies it in the prefix-shared Attention.

8 CONCLUSION

This paper presents BatchLLM, the holistic optimization techniques for large batched LLM-based information processing and management tasks. It identifies the limitations of existing methods, and optimizes the tasks with the global information of the large batch. It explicitly extracts the common prefix globally to avoid prefix's early eviction problem, and simplifies the prefix pattern by enlarging the first-level prefix with a DP algorithm on the tree to reduce the overhead of scheduling and Attention computation. It schedules the requests at the granularity of prefix-sharing groups, which enables the global prefix sharing the best and shrinks the lifetime of prefix's KV memory. It proposes the request reordering and memory-centric token batching method to better mix the prefill chunks into the decoding token-batches and thus better saturates the GPU. Finally, it presents the horizontal fused prefix-shared Attention kernel to reduce the tail effect and kernel launch overhead. Extensive evaluation shows that BatchLLM outperforms vLLM by 1.1× to 2.0× on a set of microbenchmarks and two typical industry workloads.

REFERENCES

[1] 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. *CoRR* abs/2308.16369 (2023).

[2] AI@Meta. 2024. Llama 3 Model Card. (2024). https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md

[3] 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. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023.* Association for Computational Linguistics, 4895–4901.

[4] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. 2024. Medusa: Simple LLM Inference Acceleration Framework with Multiple Decoding Heads. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024.* OpenReview.net. https://openreview.net/forum?id=PEpbUobfJv

[5] Zui Chen, Lei Cao, and Sam Madden. 2023. Lingua Manga: A Generic Large Language Model Centric System for Data Curation. *Proc. VLDB Endow.* 16, 12 (Aug. 2023), 4074–4077. https://doi.org/10.14778/3611540.3611624

[6] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

[7] Raul Castro Fernandez, Aaron J. Elmore, Michael J. Franklin, Sanjay Krishnan, and Chenhao Tan. 2023. How Large Language Models Will Disrupt Data Management. *Proc. VLDB Endow.* 16, 11 (July 2023), 3302–3309. https://doi.org/10.14778/3611479.3611527

[8] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2022. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers. *CoRR* abs/2210.17323 (2022).

[9] Yichao Fu, Siqi Zhu, Runlong Su, Aurick Qiao, Ion Stoica, and Hao Zhang. 2024. Efficient LLM Scheduling by Learning to Rank. *CoRR* abs/2408.15792 (2024).

[10] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. 2024. Cost-Efficient Large Language Model Serving for Multi-turn Conversations with CachedAttention. In *Proceedings of the 2024 USENIX Annual Technical Conference, USENIX ATC 2024, Santa Clara, CA, USA, July 10-12, 2024*. USENIX Association, 111–126.

[11] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. 2018. Low latency RNN inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference*. Association for Computing Machinery, Article 31, 15 pages. https://doi.org/10.1145/3190508.3190541

[12] In Gim, Guojun Chen, Seung-Seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2024. Prompt Cache: Modular Attention Reuse for Low-Latency Inference. In *Proceedings of the Seventh Annual Conference on Machine Learning and Systems, MLSys 2024, Santa Clara, CA, USA, May 13-16, 2024*. mlsys.org. https://proceedings.mlsys.org/paper_files/paper/2024/hash/a66caa1703fe34705a4368c3014c1966-Abstract-Conference.html

[13] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, and Yuxiong He. 2024. DeepSpeed-FastGen: High-throughput Text Generation for LLMs via MII and DeepSpeed-Inference. *CoRR* abs/2401.08671 (2024).

[14] 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. *CoRR* abs/2406.17565 (2024).

[15] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de Las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7B. *CoRR* abs/2310.06825 (2023).

[16] Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Xin Liu, Xuanzhe Liu, and Xin Jin. 2024. RAGCache: Efficient Knowledge Caching for Retrieval-Augmented Generation. *CoRR* abs/2404.12457 (2024).

[17] Yibo Jin, Tao Wang, Huimin Lin, Mingyang Song, Peiyang Li, Yipeng Ma, Yicheng Shan, Zhengfan Yuan, Cailong Li, Yajing Sun, Tiandeng Wu, Xing Chu, Ruizhi Huan, Li Ma, Xiao You, Wenting Zhou, Yunpeng Ye, Wen Liu, Xiangkun Xu, Yongsheng Zhang, Tiantian Dong, Jiawei Zhu, Zhe Wang, Xijian Ju, Jianxun Song, Haoliang Cheng, Xiaojing Li, Jiandong Ding, Hefei Guo, and Zhengyong Zhang. 2024. P/D-Serve: Serving Disaggregated Large Language Model at Scale. *CoRR* abs/2408.08147 (2024).

[18] Jordan Juravsky, Bradley Brown, Ryan Ehrlich, Daniel Y. Fu, Christopher Ré, and Azalia Mirhoseini. 2024. Hydragen: High-Throughput LLM Inference with Shared Prefixes. arXiv:2402.05099 [cs.LG]

[19] Sein Kim, Hongseok Kang, Seungyoon Choi, Donghyun Kim, Minchul Yang, and Chanyoung Park. 2024. Large Language Models meet Collaborative Filtering: An Efficient All-round LLM-based Recommender System. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '24)*. Association for Computing Machinery, 1395–1406. https://doi.org/10.1145/3637528.3671931

[20] 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, SOSP 2023, Koblenz, Germany, October 23-26, 2023*. ACM, 611–626.

[21] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. 2022. Automatic Horizontal Fusion for GPU Kernels. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2022, Seoul, Korea, Republic of, April 2-6, 2022*. IEEE, 14–27.

[22] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. 2024. AWQ: Activation-aware Weight Quantization for On-Device LLM Compression and Acceleration. In *Proceedings of the Seventh Annual Conference on Machine Learning and Systems, MLSys 2024, Santa Clara, CA, USA, May 13-16, 2024*, Phillip B. Gibbons, Gennady Pekhimenko, and Christopher De Sa (Eds.). mlsys.org.

[23] Elyas Meguellati, Lei Han, Abraham Bernstein, Shazia Sadiq, and Gianluca Demartini. 2024. How Good are LLMs in Generating Personalized Advertisements?. In *Companion Proceedings of the ACM Web Conference 2024 (WWW '24)*. Association for Computing Machinery, 826–829. https://doi.org/10.1145/3589335.3651520

[24] Maxim Milakov and Natalia Gimelshein. 2018. Online normalizer calculation for softmax. *CoRR* abs/1805.02867 (2018). arXiv:1805.02867 http://arxiv.org/abs/1805.02867

[25] OpenAI. Cited 2024. Introducing OpenAI o1-preview. https://openai.com/index/introducing-openai-o1-preview/.

[26] OpenAI. Cited 2024. OpenAI Prompt Caching. https://platform.openai.com/docs/guides/prompt-caching.

[27] Zaifeng Pan, Zhen Zheng, Feng Zhang, Ruofan Wu, Hao Liang, Dalin Wang, Xiafei Qiu, Junjie Bai, Wei Lin, and Xiaoyong Du. 2023. RECom: A Compiler Approach to Accelerating Recommendation Model Inference with Massive Embedding Columns. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*. ACM, 268–286.

[28] 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. *CoRR* abs/2407.00079 (2024).

[29] Markus N. Rabe and Charles Staats. 2021. Self-attention Does Not Need O(n^2) Memory. *CoRR* abs/2112.05682 (2021). arXiv:2112.05682 https://arxiv.org/abs/2112.05682

[30] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E. Gonzalez, and Ion Stoica. 2024. Fairness in Serving Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*. USENIX Association, 965–988.

[31] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA (Proceedings of Machine Learning Research)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.), Vol. 202. PMLR, 31094–31116.

[32] TensorRT-LLM team. Cited 2024. TensorRT-LLM. https://github.com/NVIDIA/TensorRT-LLM.

[33] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2019)*. Association for Computing Machinery, 10–19.

[34] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008.

[35] Liang Wang, Nan Yang, Xiaolong Huang, Linjun Yang, Rangan Majumder, and Furu Wei. 2023. Large Search Model: Redefining Search Stack in the Era of LLMs. *SIGIR Forum* 57, 2 (2023), 23:1–23:16.

[36] Haojun Xia, Zhen Zheng, Yuchao Li, Donglin Zhuang, Zhongzhu Zhou, Xiafei Qiu, Yong Li, Wei Lin, and Shuaiwen Leon Song. 2023. Flash-LLM: Enabling Low-Cost and Highly-Efficient Large Generative Model Inference With Unstructured Sparsity. *Proc. VLDB Endow.* 17, 2 (2023), 211–224.

[37] Haojun Xia, Zhen Zheng, Xiaoxia Wu, Shiyang Chen, Zhewei Yao, Stephen Youn, Arash Bakhtiari, Michael Wyatt, Donglin Zhuang, Zhongzhu Zhou, Olatunji Ruwase, Yuxiong He, and Shuaiwen Leon Song. 2024. Quant-LLM: Accelerating the Serving of Large Language Models via FP6-Centric Algorithm-System Co-Design on Modern GPUs. In *Proceedings of the 2024 USENIX Annual Technical Conference, USENIX ATC 2024, Santa Clara, CA, USA, July 10-12, 2024*. USENIX Association, 699–713.

[38] Haoyi Xiong, Jiang Bian, Yuchen Li, Xuhong Li, Mengnan Du, Shuaiqiang Wang, Dawei Yin, and Sumi Helal. 2024. When Search Engine Services meet Large Language Models: Visions and Challenges. *CoRR* abs/2407.00128 (2024).

[39] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zhihao Fan. 2024. Qwen2 Technical Report. *arXiv preprint*

arXiv:2407.10671 (2024).

[40] Lu Ye, Ze Tao, Yong Huang, and Yang Li. 2024. ChunkAttention: Efficient Self-Attention with Prefix-Aware KV Cache and Two-Phase Partition. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*. Association for Computational Linguistics, 11608–11620.

[41] 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

[42] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*. USENIX Association, 521–538.

[43] 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). *IEEE Transactions on Knowledge and Data Engineering* 36, 11 (2024), 6889–6907. https://doi.org/10.1109/TKDE.2024.3392335

[44] 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. 2023. Efficiently Programming Large Language Models using SGLang. (2023).

[45] Lei Zhu, Xinjiang Wang, Wayne Zhang, and Rynson W. H. Lau. 2024. RelayAttention for Efficient Large Language Model Serving with Long System Prompts. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*. Association for Computational Linguistics, 4945–4957.