BATCHLLM: OPTIMIZING LARGE BATCHED LLM INFERENCE WITH GLOBAL PREFIX SHARING AND THROUGHPUT-ORIENTED TOKEN BATCHING

Zhen Zheng [1] **Xin Ji** [1] **Taosong Fang** [1 2] **Fanghao Zhou** [1] **Chuanjie Liu** [1] **Gang Peng** [1]

ABSTRACT

Large language models (LLMs) increasingly play an important role in a wide range of information processing and management tasks in industry. Many of these tasks are performed in large batches or even offline, and the performance indicator 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 be prematurely evicted with the implicit cache management. Besides, the streaming oriented systems do not leverage the request-batch information and can not mix the decoding tokens with the prefill chunks to the best for the batched scenarios, and thus fails to saturate the GPU. 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. 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. Extensive evaluation shows that BatchLLM outperforms vLLM and SGLang by $1.3\times$ to $10.8\times$ on a set of microbenchmarks and a typical industry workload under different hardware environments. Code is available at https://github.com/microsoft/MixLLM/tree/batchllm_vllm_064.

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 (Wang et al., 2023; Xiong et al., 2024), advertisement (Meguellati et al., 2024), and recommender systems (Zhao et al., 2024; Kim et al., 2024). 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 (Kwon et al., 2023; Zheng et al., 2023) can flexibly support the execution of LLM inference tasks, especially with blocked KV memory management (Kwon et al., 2023) and per-iteration token batching (Yu et al., 2022). 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 engines support 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

[1]Microsoft [2]Chinese Information Processing Laboratory, ISCAS, UCAS. Work was done when Taosong Fang was interned at Microsoft. Correspondence to: Zhen Zheng <zhengzhen@microsoft.com>.

Proceedings of the 9^{th} MLSys Conference (Industry Track), Bellevue, WA, USA, 2026.

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 GPUs from saturating for the iterations dominated by decoding tokens.

In this work, we introduce BatchLLM, the holistic optimization for the throughput-oriented large-batch LLM inference with global prefix preprocessing, and prefix-aware and throughput-oriented token batching. 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[1]. It 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, and proposes the requests reordering and memory-centric token batching method to increase the per-iteration GPU utilization.

► It presents BatchLLM implementation building upon vLLM. BatchLLM achieves end-to-end performance improvement of $1.3\times$ to $10.8\times$ than vLLM and SGLang for the benchmarks and an industry workload under different hardware environments.

2 OPPORTUNITIES

2.1 Emerging Industry Scenarios

LLM has been increasingly used in a broad range of industry tasks with its in-context learning and reasoning ability (Wang et al., 2023; Xiong et al., 2024; Meguellati et al., 2024; Zhao et al., 2024; Kim et al., 2024; Chen et al., 2023; Fernandez et al., 2023). 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 (Wang et al., 2023; Xiong et al., 2024) 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 industry (Wang et al., 2023) 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, a document can be the common prefix for different queries. There are many other industry tasks that fall into the offline processing mode with a common prefix. The very important offline ranking tasks for search and recommendation can use the prompt of *"please evaluate the relevance of the document for the query, Query: <...>, Document: <...>"*, where each query corresponds to many documents and can be the common prefix. The Ads title rewrite task in industry can use the prompt of *"given the Query and the Landing Page info, please refine the old title 'xxx' to get a better ad title. Landing Page: <...>, Query: <...>"*, where the landing page can be the common prefix and the computation is usually throughput critical. The offline labeling and scoring tasks in industry can have long system prompt and few shot examples, which can also be the common prefix in the prompt.

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 instructions becomes shorter and shorter: On the one hand, the LLM models are incorporating more and more reasoning abilities into themselves (like OpenAI o1 (OpenAI, Cited 2024a)) 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 instructions can be the most important for many industry tasks. This motivates the first-level prefix enlargement in Sec.3.2.

[1]This paper uses the term *token-batch* to represent a batch of tokens to be processed for an iteration of continuous batching (Yu et al., 2022). It differs from the concept of *large batch* of all the prompt to be processed in a job.

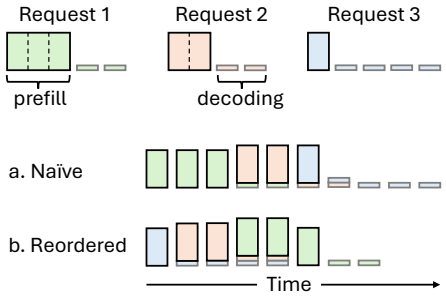


Figure 1: The effect of processing order of requests with chunked-prefill enabled. Given the three requests with different prefill/decoding length characteristics, the naive token batching in the coming order of the requests has worse token mixing of decoding and prefill chunks.

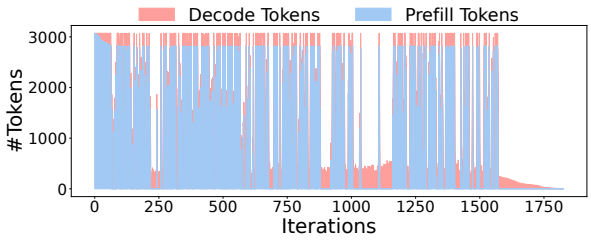


Figure 2: The token number in the batch processed at each iteration for an industry task with vLLM's chunked-prefill. It has "valleys" for many iterations.

2.2 The New Optimization Demands

Even though prefix sharing and per-iteration token batching are functionally supported in previous works, they lack specialized optimization for the increasingly import large batched scenarios (Sec.2.1), showing two main limitations:

1. The implicit prefix caching cannot achieve the optimal KV reuse for the large batched inference. The industrial LLM serving systems (Kwon et al., 2023; Zheng et al., 2023) use the prefix tree to maintain the KV blocks. The same prefix (identified by runtime hashing) can be 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 58.1% by analyzing the input dataset. Note that the saving ratio is calculated by:

$$R_{saving} = (1 - \frac{N_{processed\_pre\_tok}}{N_{logical\_pre\_tok}}) \times 100 \qquad (1)$$

$N_{processed\_pre\_tok}$ is the number of prefill tokens processed after prefix sharing, and $N_{logical\_pre\_tok}$ is the original prefill token number of all the requests. The vLLM's implicit prefix caching only achieves 35.8% (Sec.4.3.1). There is a need for better prefix sharing for the large batched tasks.

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 (Holmes et al., 2024; Agrawal et al., 2023) of other requests. 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 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.4.2.2) performing with vLLM's best configuration. It shows that there are "valleys" at many iterations where the number of tokens are not 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 Design Methodology

3.1 Overview

BatchLLM has the following insights to address the limitations described in Sec.2.2.

1. Explicit prefix identification and sharing. Instead of managing 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.

2. Grouped scheduling, request reordering and memory-

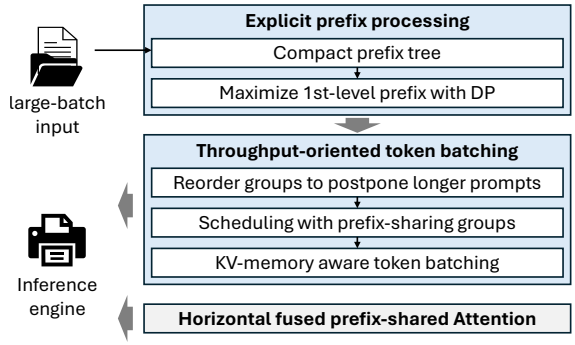


Figure 3: BatchLLM overview.

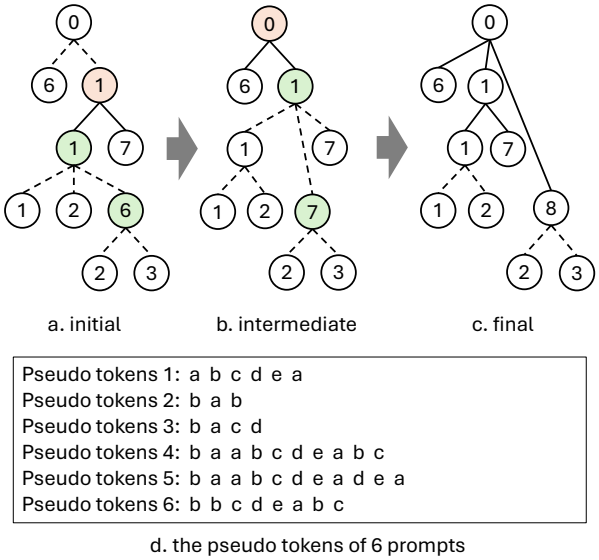


Figure 4: The preprocessing to maximize the first level prefix reusing. Each circle represents a node containing a number of tokens. The number in the circle is the token number of the node.

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 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".

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 industry tasks are usually dominated by a long context (Sec.2.1). The explicit prefix processing is illustrated in Sec.3.2. The requests will be organized into groups where each group corresponds to the queries 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 postpone the groups with longer prefill. 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.3.3.

3.2 Explicit Global KV Reuse Identification

As discussed in Sec.2.1, 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 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, based on the insight that the length of the prefixes are usually dominated by a single level (Sec.2.1).

BatchLLM explicitly identifies the common prefix among the prompts within the large batch. It makes use of the compact prefix tree[2] 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 pseudo 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 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.

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

[2]The compact prefix tree, also called Radix tree, is the prefix tree that merges the only child with its parent.

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

```
1: ▷ The 1st-level prefixes of a node are its children
2: function MaximizeReuse(D)
3:   if D has no children then
4:     return
5:   for child ∈ D.children do
6:     MaximizeReuse(child)        ▷ Solve sub-problems
7:     for gchild ∈ child.children do
8:       gain ← (leaves(gchild) - 1) × tokens(gchild)
9:       penalty ← tokens(child)
10:      if gain > penalty then
11:        Fork child and merge with gchild
```

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 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.1. 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.4.2.2). 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. More details will be presented in Sec.4.3.1.

Time and Space Complexity Analysis. The complexity of Algo.1 is $O(nc^2)$, where n is the number of nodes and c is the number of average children. The algorithm visits every node once and has $O(n)$ for traversal, and iterates over its children and their children for each node and has $O(c^2)$ per node as the worst case. Thus, the time complexity is $O(nc^2)$. In real scenarios, c is typically small on average, so this is close to $O(n)$ in practice. The space complexity is $O(n)$. The memory overhead introduced by "fork child and merge with grandchild" is very small, as the metadata of the node is the `token_id`, which is in `int64` datatype. The worst case is that all nodes are fully forked, which means the same prefix of different prompts are stored independently. In this case, the storage will be equal to the sum of token numbers of all prompts (in `int64` datatype), which is still small compared to the memory consumption of model weights. Algo.1 does not introduce KV residency time as it is computed based on the input `token_ids` before processing the tokens. As a result, the DP algorithm only introduces less than 0.01% overhead in our evaluation (Sec.4.3.1)

BatchLLM does not completely disable the LRU mechanism. It explicitly applies global prefix cache to the longest first-level prefix and relies on vLLM's LRU to manage the possible remaining prefixes (e.g., the second-level sub-prefix) for distinct prompt parts.

3.3 Throughput-oriented Token Batching

3.3.1 Prefix-sharing Group Based Scheduling

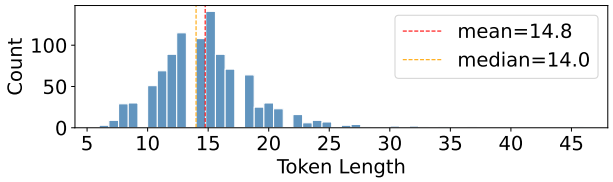
To simplify the KV reuse and reduce the lifetime of the common prefix in the memory, BatchLLM schedules the requests sharing the same prefix all together. Specifically, the procedure described in Sec.3.2 will generate the *prefix-sharing groups* (the collection of requests sharing the same prefix) and BatchLLM will schedule the requests at the granularity of *prefix-sharing group* rather than each single request. 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 based prefix management 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 (Holmes et al., 2024). 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) are scheduled before the inactive requests, which finishes the active requests earlier and releases their memory earlier.

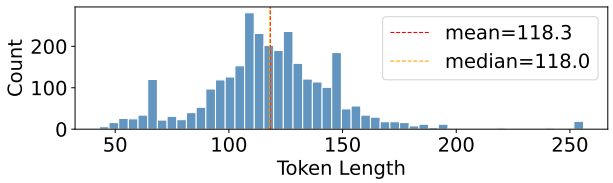
3.3.2 Request Groups Reordering

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:

$$R = \frac{L_{decode}}{L_{prefill}} \tag{2}$$



(a) Output length distribution of Ads title rewrite.



(b) Output length distribution of caption generation.

Figure 5: Generation output length distribution of two typical real-industry workloads.

In Eq.2, $L_{prefill}$ 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' 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 industry tasks (ads, caption generation, offline ranking, etc.) and observed that the distribution of output length is relatively concentrated for the same kind of task, which means the effect of the output length in Eq.2 is less significant for the same kind of task as it does not vary a lot. We thus make an approximation by normalizing L_{decode} into 1 in Eq.2 for all requests uniformly. As we schedule the requests in the unit of *prefix-sharing group*, the ratio of the group (after normalization) becomes:

$$R_{group} = \frac{1}{L_{prefix} + \sum L_{distinct}} \tag{3}$$

The request groups with larger R_{group} will be scheduled with higher priority.

Fig.5 shows the output length distributions of two representative industry workloads. It shows that the the output lengths are in a very narrow range, which motivates our choice of normalizing decoding length.

3.3.3 *Memory-centric Token Batching to Saturate GPU*

Another limiter of the size of token-batches is that, the existing works (Kwon et al., 2023) 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 (noted by the vLLM community[3]), as 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.

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. 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.

3.4 End-to-end System Integration

The integration of BatchLLM into other frameworks is quite easy. BatchLLM comprises two pluggable modules: (i) a token-batching scheduler and (ii) an attention backend, that can be added alongside existing components in common inference frameworks (e.g., vLLM, SGLang) without altering user-facing APIs or the serving loop. Token-batching scheduler is orthogonal to the framework's existing continuous batching/prefill–decode pipeline. We add lightweight per-request metadata to indicate shared prefix vs. distinct part. The scheduler ensures the common prefix across requests is computed once and reused, reducing redundant calculation in both attention (KV-cache construction) and non-attention compute (e.g., projection and FFN GEMMs) during prefill and decoding. Attention backend is a drop-in backend with full fallback. The backend presents the same public

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

interface as the framework's attention module and simply registers as an additional backend. If shapes/hardware are unsupported, it falls back to the standard kernel, ensuring compatibility and graceful degradation.

4 EVALUATION

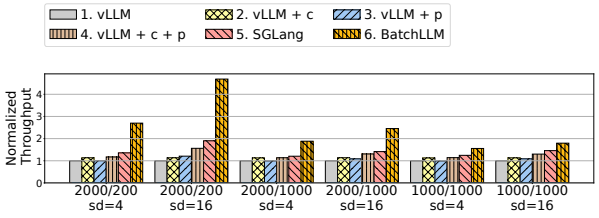
4.1 Setup

Baseline Specification. We compare BatchLLM with vLLM and SGLang, two state-of-the-art LLM inference engines with prefix sharing, for the end-to-end comparison to demonstrate the efficacy (Sec.4.2). We integrate all our design in vLLM v0.6.4 and the baseline for comparison is also vLLM v0.6.4 (except as noted).

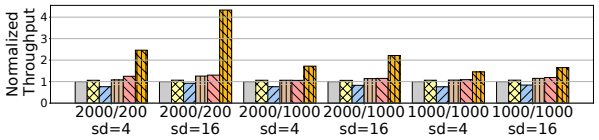
We enable the *prefix-caching* and *chunked-prefill* in vLLM baseline for the end-to-end comparison in Sec.4.2. The *prefix-caching* uses the implicit LRU-based caching policy. We use 2048 as the token-batch size for chunked-prefill of vLLM baseline to maximize its throughput.

Hardware and System Specification. We conduct our experiments on both NVIDIA A100 80GB GPUs (both single GPU and 2-GPUs) and AMD MI200 GPU. For multi-GPU experiments on NVIDIA A100, we utilize tensor parallelism (TP) with size of 2 (denoted as A100-TP2). 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.

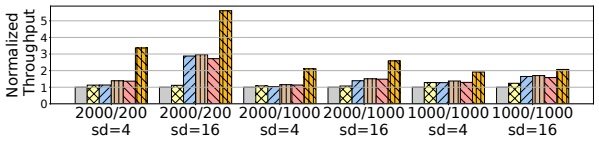
Models and Dataset. We evaluate BatchLLM and the baselines with Llama-3.1-8B (AI@Meta, 2024), Qwen-2.5-14B (Qwen et al., 2025) and Llama-3.1-70B (4-bit GPTQ), with a set of input/output with different data distributions, and Qwen-2.5-7B for a typical industry task. As for the basic evaluation, we use three different settings of the length of common prefix and distinct prompt. The length of prefix/distinct are 2000/200, 2000/1000 and 1000/1000 respectively, while the length of generated tokens is 100. We evaluate the three settings under different share-degrees. The share-degree means the number of distinct prompts sharing the same prefix. Note that all requests are randomly shuffled. The above input/output settings are very representative as they come from the real industry workloads running on thousands of GPUs. We use a uniform length to avoid analysis inconvenience caused by different lengths. Additionally, to assess performance under more dynamic lengths, we conduct experiments on Llama-3.1-8B (A100, TP=1) where the lengths of the shared prefix, distinct context, and generated tokens are perturbed around base values using a Poisson distribution. These experiments help evaluate the robustness of BatchLLM and baselines to variations in sequence lengths. The microbenchmark evaluation is in Sec.4.2.1. As for the industry tasks, we conduct the evaluation on a typical scenario in web search engines, which is the snippet



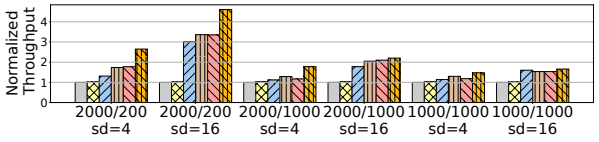
(a) Llama 3.1 8B throughput on A100, #requests = 6400



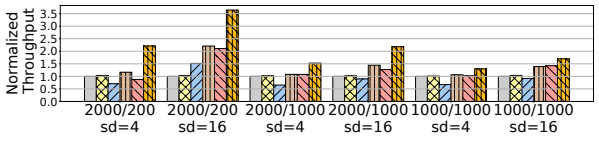
(b) Qwen 2.5 14B throughput on A100, #requests = 6400



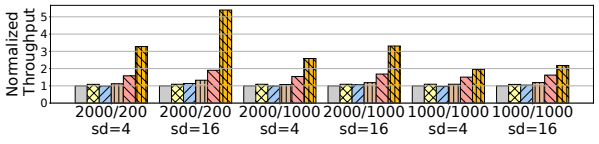
(c) Llama 3.1 70B throughput, on A100, #requests = 800



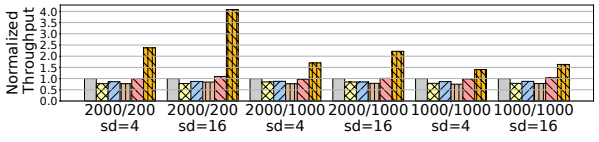
(d) Llama 3.1 70B throughput, on A100-TP2, #requests = 800



(e) Llama 3.1 8B throughput with Poisson perturbed lengths on A100, #requests = 6400



(f) Llama 3.1 8B throughput, on MI200, #requests = 6400



(g) Qwen 2.5 14B throughput, on MI200, #requests = 6400

Figure 6: Microbenchmark evaluation. The setting *m/n* (like *2000/200*) indicates the length of shared prefix/non-shared context, *sd* means *sharing degree*. For subfigure (e), lengths are perturbed around base values using a Poisson distribution. The vLLM setting with '+ p' ('+ c') means *prefix-caching* (*chunked-prefill*) enabled.

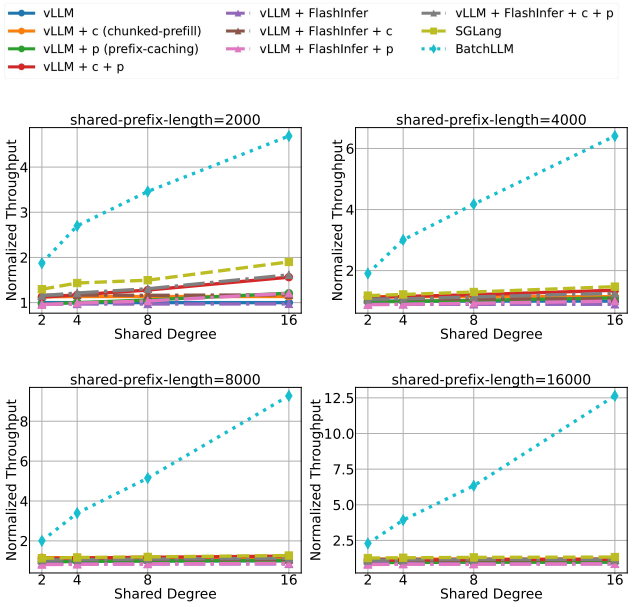


Figure 7: Microbenchmark evaluation of different sharing-degrees and different shared-prefix lengths on A100. Given 6400 requests. This evaluation is based on Llama-3.1-8B-Instruct, while the length of non-shared context is 200, and the generated length is 100.

generation task that generates the snippet of a document for the user query. We conduct the breakdown experiments in Sec.4.3 to demonstrate the efficacy of each technique.

4.2 End to End Comparison

4.2.1 Basic Evaluation

Fig.6 shows the end-to-end throughput comparison between BatchLLM and the baselines with different shared-prefix/non-shared context settings. The results show that BatchLLM outperforms the vLLM and SGLang 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*, which performs the best across all vLLM's configurations, BatchLLM can get 1.4-3× speedup on A100 and 1.8-4.1× speedup on MI200. BatchLLM also achieves better performance than SGLang, with 1.2-2.5× speedup on A100 and 1.3-2.8× speedup on MI200. The acceleration is stable from 7B to 70B models.

The impact of sharing-degree and shared prefix. We also compare the throughput of different settings after changing the sharing-degree and the length of shared prefix. Fig.7 shows that the speedup becomes more significant when more requests share the same prefix and the portion of the common prefix becomes larger, which contributes to a bigger reuse ratio, and BatchLLM can save more computation and memory for this case. With lower sharing-degree and shorter common prefix (like *sd* is 2 and the length of prefix is 2000), the acceleration of BatchLLM is 1.7×/1.5×, comparing with the best setting of vLLM and SGLang. When the length of shared prefix is 16000, and the share degree is 16, the best performance of vLLM and SGLang are 0.49 and 0.56, while BatchLLM achieves 10.8×/9.5× speedup.

Table 1: Industry workload evaluation.

Settings	Throughtput, A100	Throughput, MI200
vLLM	5.45	2.75
+ c (**c**hunked-prefill)	5.48	2.84
+ p (**p**refix-caching)	6.2	3.27
+ c + p	6.71	3.36
BatchLLM	**8.67**(1.30×)	**4.24**(1.26×)

Comparison between vLLM + request sorting and BatchLLM. We perform a preprocess for the setting of *vLLM + chunked-prefill + prefix-cahcing*: use *python.sorted()* to gather all requests with the same prefix together[4]. This can help vLLM to cache the tokens better. After this preprocess, vLLM's throughput is boosted from 6 to 13.02, while the throughput of BatchLLM is 18. One reason is that vLLM cannot simplify the attention calculation when there are more than 1 common prefix in the batch every step of its inference, while BatchLLM could handle it. Besides, BatchLLM has a better token-batching strategy than vLLM.

Comparison with the latest prefix-caching advances by early 2026. BatchLLM is implemented based on an old vLLM version, v0.6.4. The latest vLLM version has integrated many of the most recent prefix-sharing technologies from the community, and we make a comparison against it. We evaluate the industry workload in this paper with the latest vLLM (the vLLM's github commit by 1/16/2026). Its throughput on A100 is 6.57 for this workload with its latest prefix-caching optimization, while BatchLLM achieves 8.67 even with a very old vLLM as the implementation base.

Overall, 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.4.3.

4.2.2 Industry Workload Evaluation

This industry workload is similar to the snippet generation task described in the recent work (Wang et al., 2023), which is also described in Sec.2.1. The original input has two levels of prefixes: the global shared instruction to extract the information from the document-query pair, and the shared

[4]Here the length of shared prefix/non-shared context/generated tokens are 2000/200/100, while the sharing degree is 16. The model is Llama-3.1-8B. And the GPU is NVIDIA A100.

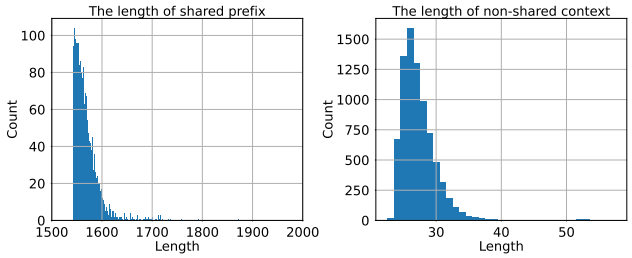


Figure 8: The length distribution of shared prefix/non-shared context in the dataset of the industry workload.

document between different groups of queries. The first-level prefix is very short. With the optimization described in Sec.3.2, 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.1. 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. Besides, the DP algorithm only introduces less than 0.01% overhead.

The data distribution of this workload is shown in Fig.8. The average share-degree (i.e., the number of the distinct prompt that share the same prefix) is about 3, the average common prefix length is about 1570 tokens, and the average distinct prompt length is about 30 tokens. We sample 8000 queries for this case study.

We conduct this experiment on the NVIDIA A100 GPU and AMD MI200 GPU. The model is a fine-tuned Qwen-2.5-7B model. Table.1 shows the effectiveness of BatchLLM of this workload, with about 1.3×/1.26× speedup over the best configuration of vLLM. BatchLLM better reuses the KV context than the vLLM baseline with the optimization in Sec.3.2, for which we have the breakdown analysis in Sec.4.3.1. BatchLLM also better mixes the decoding tokens with prefill tokens to increase the size of token-batches with the optimization in Sec.3.3, for which we have the breakdown analyze in Sec.4.3.2. It also benefits from the optimized Attention kernel described later. During the execution, it does not have KV swaps/recomputations in BatchLLM when processing the industry workload above. The peak KV usage is about 50% for this workload.

4.2.3 Heavy-tailed decoding with varied output lengths.

We evaluate heavy-tailed decoding to demonstrate BatchLLM's efficacy for varied output lengths, using a Pareto-distributed output length configuration.

We generate a workload of 320 requests sampled from a Pareto distribution (shape parameter $\alpha = 1.5$) bounded between 10 and 512 tokens. This distribution yields a mean generation length of 51.2 tokens. However, the heavy-tail nature of the workload is best illustrated by its percentiles: while the median ($p50$) is just 25 tokens, the 99th percentile ($p99$) reaches 509 tokens, resulting in an extreme $p99/p50$ ratio of 20.3×. The evaluation is conducted using the Qwen 2.5 7B model at FP16 precision. To stress-test the prefix-sharing capabilities of our system, the 320 requests are divided into 20 groups. Each group consists of 16 requests that share a common, lengthy prefix of 2000 tokens, appended with a distinct prompt of 200 tokens (totaling an input length of 2200 tokens per request). We enable chunked prefill with a maximum of 2048 batched tokens and configure a KV cache block size of 64.

Table 2: End-to-end throughput under the heavy-tail workload.

Configuration	Throughput (req/s)	Speedup
vLLM	5.26	1.0×
BatchLLM	17.02	**3.2**×

We compare BatchLLM with vLLM. Tab.2 summarizes the end-to-end throughput under the heavy-tail workload. BatchLLM achieves a substantial 3.2× speedup over the baseline, increasing throughput from 5.26 req/s to 17.02 req/s. This performance gain demonstrates that BatchLLM's context sharing mechanism is highly resilient to the decoding inefficiencies traditionally caused by heavy-tail distributions.

4.3 Breakdown Analysis

To better isolate the performance improvements of BatchLLM, we provide the breakdown analysis to evaluate the individual effects of our core techniques: prefix grouping, request reordering (token-batching), and kernel fusion.

4.3.1 Effect of Prefix Grouping

Algorithm Effectiveness on Multi-level Prefix Inputs. To demonstrate the effectiveness of our proposed prefix identification algorithm, we conduct an ablation study comparing it against a baseline sorting-based heuristic approach for multi-level prefix inputs. The heuristic method relies on lexicographical sorting and greedy scanning, grouping requests only if their longest common prefix exceeds a rigid 50% length threshold relative to the shortest member. In contrast, our approach automatically discovers shared prefixes of arbitrary lengths without rigid ratio constraints, merging paths based on a dynamic gain formula.

We construct a synthetic dataset to evaluate edge cases where prompts share significant context but fall short of heuristic thresholds. Each prompt consists of 1000 tokens formulated as: $group\_prefix$ + sub_prefix +

$random\_suffix$. We generate 6400 prompts divided into 50 global groups. Each group contains 64 subcategories, with 2 prompts per subcategory. We test two configurations: Setting A uses a 490-token $group\_prefix$ and an 11-token sub_prefix, while Setting B uses a 400-token $group\_prefix$ and a 101-token sub_prefix.

In both settings, the prefix ratio within the same subcategory is $> 50\%$ (e.g., $(490 + 11)/1000 = 50.1\%$), but the prefix ratio across different subcategories within the same global group falls strictly below 50% (e.g., $490/1000 = 49\%$).

Table 3: Performance comparison of prefix grouping algorithms across 6400 requests. The baseline executes without context sharing (CS) or BatchLLM optimizations.

Configuration	Saving Ratio	Throughput (req/s)
Setting A ($group\_prefix = 490$)		
Baseline (No CS)	0.0%	10.7
Heuristic CSGroup	∼25.0%	12.4
BatchLLM	**48.6%**	**15.9**
Setting B ($group\_prefix = 400$)		
Baseline (No CS)	0.0%	10.7
Heuristic CSGroup	∼25.0%	12.3
BatchLLM	**39.7%**	**14.5**

As shown in Table 3, the heuristic approach is bottlenecked by its 50% threshold. It successfully groups prompts within the same subcategory (forming 3200 groups of 2 members), yielding a limited token saving ratio of ∼25% and throughput of ∼12.4 prompts/s.

Conversely, our longest prefix identification algorithm completely bypasses this limitation, identifying the global $group\_prefix$ and clustering all 128 members of a group together (forming 50 groups). In Setting A, BatchLLM achieves a 48.6% token saving ratio and a throughput of 15.9 prompts/s, delivering a 1.28× speedup over the heuristic method and a 1.49× speedup over the baseline. Furthermore, BatchLLM grouping closely matches theoretical ground truth grouping limits, proving its robustness in capturing complex, multi-level prefix hierarchies.

KV Cache Reusing Effect. To show that this grouping optimizes the KV reuse, we evaluate the throughput and the token saving ratio in both microbenchmarks and the industry task. Note that all samples in the the industry task are not sorted too.

As shown in Fig.9, we can see the token saving ratio is improved in BatchLLM across different settings. And it becomes higher when the sharing-degree is larger. Specifically, in the microbenchmark setting where length of shared prefix is 2000 and the *sharing-degree* is 16, BatchLLM achieves 85.2% token saving ratio, the upper bound in this scenario, comparing to the 40.1% token saving ratio in *vLLM + chunked-prefill + prefix-caching*. With the increase

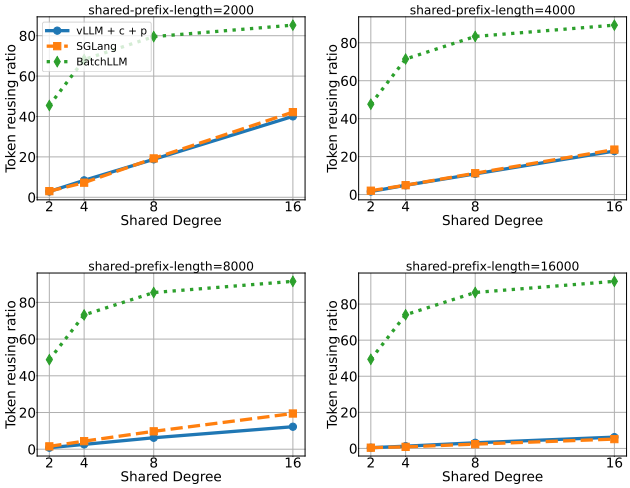


Figure 9: The token saving (or reusing) ratio of different sharing-degrees and different shared-prefix lengths on A100, based on the microbenchmark. This evaluation is based on Llama-3.1-8B-Instruct, while the length of non-shared context is 200, and the generated length is 100.

of the requests number and the length of prefix, it can be expected that the implicit LRU cache policy of vLLM's prefix-caching can suffer more from the eviction of the reusable KV context, while the inference could benefit more from BatchLLM. For example, when the length of shared prefix is 16000, and the share degree is 16, the token reusing ratio of vLLM and SGLang are 6.3% and 5.2%, while the reusing ratio of BatchLLM is 92.6 %, much better than the other two baselines.

Global Prefix Identification Overhead. We have measured the time of the global prefix identification described in Sec.3.2. 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.4.2.2, the total overhead of the 8000 requests is 1.51 seconds. While the time to process the 8000 requests with the input of prefix-sharing groups format is 919.54 seconds. The global prefix identification procedure nearly has no overhead compared with the end-to-end execution time.

4.3.2 *Effect of Request Reordering and Token-Batching*

Fig.10 shows the number of per-iteration tokens, using the same workload with Fig.2. It clearly shows that the request reordering and token-batching optimization in BatchLLM successfully reduces the "valleys" of the iterations, thus increasing the overall GPU utilization.

We evaluated our token-batching optimization through ablation studies on throughput-oriented token-batching. Testing on Qwen-2.5-7B with 3000 industry workload query (Fig.4)

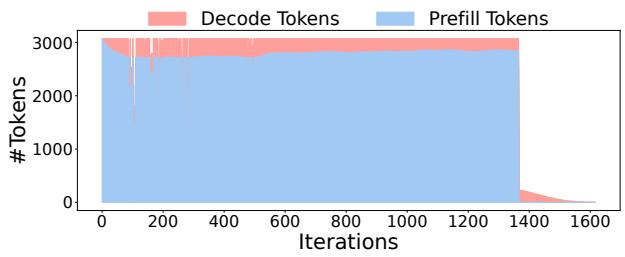


Figure 10: Token number per iteration with token-batching optimization (Sec.3.3), showing significant valley reduction compared to Fig.2's baseline. Prefix-sharing disabled to demonstrate general batched/offline inference efficacy.

Table 4: Token-Batching Breakdown.

Settings	Disable token-batching	Enable token-batching
vLLM + c	5.79	**5.94**
BatchLLM	8.16	**8.47**

shows effective performance improvement. Results demonstrate effectiveness both with and without prefix-sharing, with combined optimizations working particularly well in prefix-sharing scenarios.

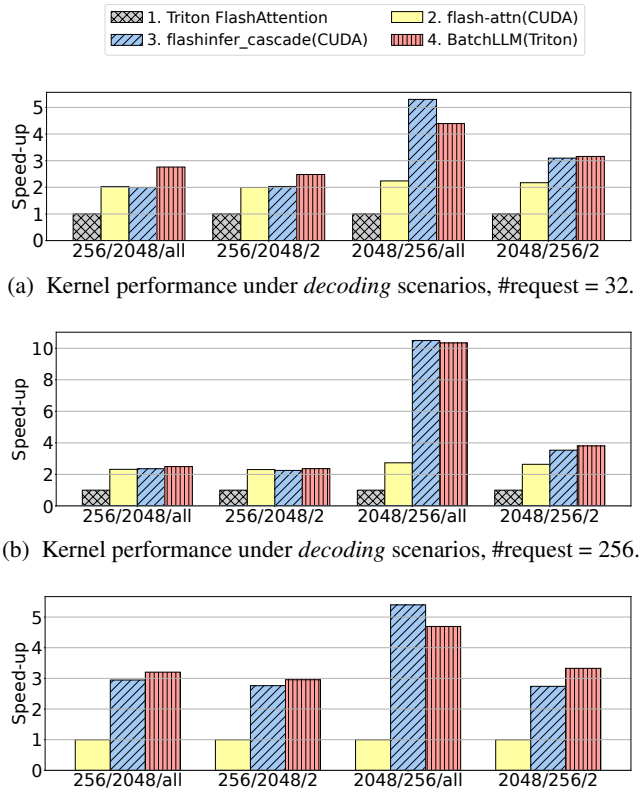
4.3.3 *Effect of Horizontal Kernel Fusion*

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.

We implement the Attention in BatchLLM through OpenAI Triton language (Tillet et al., 2019). 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[5] on AMD GPUs.

We compare BatchLLM's prefix-shared Attention implementation with several baselines on both NVIDIA A100 GPU and AMD MI200 GPU. Fig.11 shows the performance comparison of different implementations, including Triton official FlashAttention implementation, official FlashAttention v2.6.1, FlashinferCascade v0.1.4 (Ye et al., 2024b), and BatchLLM's kernels. We compare the performance

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



(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 11: The performance comparison between the baseline, CascadeInfer and BatchLLM. 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* in one *prefix-sharing group*.

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).

Results show 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. This indicates the high-level Triton compiler still have a room to achieve the performance of low-level CUDA compiler for complex computations.

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.

4.4 Limitations and Deployment Considerations

While BatchLLM demonstrates superior performance in the industrial scenarios detailed in Sec.2.1, it is subject to specific limitations. Primarily, the system's design may adversely affect individual request latency, rendering it less suitable for latency-critical workloads. A potential mitigation is the development of a hybrid architecture that dynamically toggles between streaming and batching modes. Specifically, when the system encounters a backlog of tasks—transitioning into a batching-intensive regime—and identifies a high global prefix sharing ratio, it could employ BatchLLM techniques to maximize throughput and reduce queue depth. Otherwise, the system should utilize standard execution paths to prioritize low latency. We leave the implementation of such a hybrid system for future work. Secondly, the advantages of BatchLLM are diminished in environments lacking common prefixes. In the absence of prefix caching, performance gains are restricted solely to those provided by token batching. Therefore, we recommend deploying BatchLLM specifically for prefix-heavy, throughput-oriented workloads rather than general-purpose tasks.

5 Related Work

Prefix sharing systems. The vLLM (Kwon et al., 2023) 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 (Gim et al., 2024) and SGLang (Zheng et al., 2023) 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 (Jin et al., 2024a) studies the common KV caching for RAG optimization. Some recent works (Gao et al., 2024; Jin et al., 2024b; Qin et al., 2024; Hu et al., 2024; OpenAI, Cited 2024b) study the distributed request scheduling with the consideration of prompt prefix reuse. None of these works identify the common prefix ahead-of-time 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 (Yu et al., 2022) proposes the continuous batching method for transformer models with the per-iteration token batching of autoregressive models (Gao et al., 2018). FastGen (Holmes et al., 2024) and SARATHI (Agrawal et al., 2023) study the mix of prefill chunks and decoding tokens into the same token-batch to increase the compute intensity of the batch. FlexGen (Sheng et al., 2023) proposes the swizzled token computation and offloading to support the LLM inference with limited GPU memory. Some works (Fu et al., 2024; Sheng et al., 2024) 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 (Kwon et al., 2023) 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 (Xia et al., 2023; Frantar et al., 2022; Lin et al., 2024), which can be applied concurrently with the optimizations in this paper.

Attention kernel optimization. Memory-efficient Attention (Rabe & Staats, 2021) and FlashAttention (Dao et al., 2022) fuse the Attention operators into a single kernel to boost the performance, with Online Softmax (Milakov & Gimelshein, 2018) to tile the Attention computation. The recent prefix-shared Attention optimization (ChunkAttention (Ye et al., 2024a), RelayAttention (Zhu et al., 2024), Hydragen (Juravsky et al., 2024), and Cascade Inference (Ye et al., 2024b)) 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 through Online Softmax. 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 (Pan et al., 2023; 2024; Li et al., 2022). BatchLLM borrows this idea and applies it in the prefix-shared Attention.

6 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 and SGLang by $1.3\times$ to $10.8\times$ on a set of microbenchmarks and a typical industry workload under different hardware environments.

REFERENCES

Agrawal, A., Panwar, A., Mohan, J., Kwatra, N., Gulavani, B. S., and Ramjee, R. SARATHI: efficient LLM inference by piggybacking decodes with chunked prefills. *CoRR*, abs/2308.16369, 2023.

AI@Meta. Llama 3 model card. 2024. URL https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md.

Chen, Z., Cao, L., and Madden, S. Lingua manga: A generic large language model centric system for data curation. *Proc. VLDB Endow.*, 16(12):4074–4077, August 2023. ISSN 2150-8097. doi: 10.14778/3611540.3611624.

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. 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*, 2022.

Fernandez, R. C., Elmore, A. J., Franklin, M. J., Krishnan, S., and Tan, C. How large language models will disrupt data management. *Proc. VLDB Endow.*, 16 (11):3302–3309, July 2023. ISSN 2150-8097. doi: 10.14778/3611479.3611527.

Frantar, E., Ashkboos, S., Hoefler, T., and Alistarh, D. GPTQ: accurate post-training quantization for generative pre-trained transformers. *CoRR*, abs/2210.17323, 2022.

Fu, Y., Zhu, S., Su, R., Qiao, A., Stoica, I., and Zhang, H. Efficient LLM scheduling by learning to rank. *CoRR*, abs/2408.15792, 2024.

Gao, B., He, Z., Sharma, P., Kang, Q., Jevdjic, D., Deng, J., Yang, X., Yu, Z., and Zuo, P. 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*, pp. 111–126. USENIX Association, 2024.

Gao, P., Yu, L., Wu, Y., and Li, J. Low latency rnn inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference*. Association for Computing Machinery, 2018. ISBN 9781450355841. doi: 10.1145/3190508.3190541.

Gim, I., Chen, G., Lee, S., Sarda, N., Khandelwal, A., and Zhong, L. 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, 2024. URL https://proceedings.mlsys.org/paper_files/paper/2024/hash/a66caa1703fe34705a4368c3014c1966-Abstract-Conference.html.

Holmes, C., Tanaka, M., Wyatt, M., Awan, A. A., Rasley, J., Rajbhandari, S., Aminabadi, R. Y., Qin, H., Bakhtiari, A., Kurilenko, L., and He, Y. Deepspeed-fastgen: High-throughput text generation for llms via MII and deepspeed-inference. *CoRR*, abs/2401.08671, 2024.

Hu, C., Huang, H., Hu, J., Xu, J., Chen, X., Xie, T., Wang, C., Wang, S., Bao, Y., Sun, N., and Shan, Y. Memserve: Context caching for disaggregated LLM serving with elastic memory pool. *CoRR*, abs/2406.17565, 2024.

Jin, C., Zhang, Z., Jiang, X., Liu, F., Liu, X., Liu, X., and Jin, X. Ragcache: Efficient knowledge caching for retrieval-augmented generation. *CoRR*, abs/2404.12457, 2024a.

Jin, Y., Wang, T., Lin, H., Song, M., Li, P., Ma, Y., Shan, Y., Yuan, Z., Li, C., Sun, Y., Wu, T., Chu, X., Huan, R., Ma, L., You, X., Zhou, W., Ye, Y., Liu, W., Xu, X., Zhang, Y., Dong, T., Zhu, J., Wang, Z., Ju, X., Song, J., Cheng, H., Li, X., Ding, J., Guo, H., and Zhang, Z. P/d-serve: Serving disaggregated large language model at scale. *CoRR*, abs/2408.08147, 2024b.

Juravsky, J., Brown, B., Ehrlich, R., Fu, D. Y., Ré, C., and Mirhoseini, A. Hydragen: High-throughput llm inference with shared prefixes, 2024.

Kim, S., Kang, H., Choi, S., Kim, D., Yang, M., and Park, C. 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, pp. 1395–1406. Association for Computing Machinery, 2024. ISBN 9798400704901. doi: 10.1145/3637528.3671931.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. 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*, pp. 611–626. ACM, 2023.

Li, A., Zheng, B., Pekhimenko, G., and Long, F. 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*, pp. 14–27. IEEE, 2022.

Lin, J., Tang, J., Tang, H., Yang, S., Chen, W., Wang, W., Xiao, G., Dang, X., Gan, C., and Han, S. AWQ: activation-aware weight quantization for on-device LLM

compression and acceleration. In Gibbons, P. B., Pekhimenko, G., and Sa, C. D. (eds.), *Proceedings of the Seventh Annual Conference on Machine Learning and Systems, MLSys 2024, Santa Clara, CA, USA, May 13-16, 2024*. mlsys.org, 2024.

Meguellati, E., Han, L., Bernstein, A., Sadiq, S., and Demartini, G. How good are llms in generating personalized advertisements? In *Companion Proceedings of the ACM Web Conference 2024*, WWW '24, pp. 826–829. Association for Computing Machinery, 2024. doi: 10.1145/3589335.3651520.

Milakov, M. and Gimelshein, N. Online normalizer calculation for softmax. *CoRR*, abs/1805.02867, 2018. URL http://arxiv.org/abs/1805.02867.

OpenAI. Introducing openai o1-preview. https://openai.com/index/introducing-openai-o1-preview/, Cited 2024a.

OpenAI. Openai prompt caching. https://platform.openai.com/docs/guides/prompt-caching, Cited 2024b.

Pan, Z., Zheng, Z., Zhang, F., Wu, R., Liang, H., Wang, D., Qiu, X., Bai, J., Lin, W., and Du, X. 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*, pp. 268–286. ACM, 2023.

Pan, Z., Zheng, Z., Zhang, F., Xie, B., Wu, R., Smith, S., Liu, C., Ruwase, O., Du, X., and Ding, Y. Recflex: Enabling feature heterogeneity-aware optimization for deep recommendation models with flexible schedules. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2024, Atlanta, GA, USA, November 17-22, 2024*, pp. 41. IEEE, 2024.

Qin, R., Li, Z., He, W., Zhang, M., Wu, Y., Zheng, W., and Xu, X. Mooncake: A kvcache-centric disaggregated architecture for LLM serving. *CoRR*, abs/2407.00079, 2024.

Qwen, :, Yang, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Li, C., Liu, D., Huang, F., Wei, H., Lin, H., Yang, J., Tu, J., Zhang, J., Yang, J., Yang, J., Zhou, J., Lin, J., Dang, K., Lu, K., Bao, K., Yang, K., Yu, L., Li, M., Xue, M., Zhang, P., Zhu, Q., Men, R., Lin, R., Li, T., Tang, T., Xia, T., Ren, X., Ren, X., Fan, Y., Su, Y., Zhang, Y., Wan, Y., Liu, Y., Cui, Z., Zhang, Z., and Qiu, Z. Qwen2.5 technical report, 2025. URL https://arxiv.org/abs/2412.15115.

Rabe, M. N. and Staats, C. Self-attention does not need $o(n^2)$ memory. *CoRR*, abs/2112.05682, 2021. URL https://arxiv.org/abs/2112.05682.

Sheng, Y., Zheng, L., Yuan, B., Li, Z., Ryabinin, M., Chen, B., Liang, P., Ré, C., Stoica, I., and Zhang, C. Flexgen: High-throughput generative inference of large language models with a single GPU. In Krause, A., Brunskill, E., Cho, K., Engelhardt, B., Sabato, S., and Scarlett, J. (eds.), *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pp. 31094–31116. PMLR, 2023.

Sheng, Y., Cao, S., Li, D., Zhu, B., Li, Z., Zhuo, D., Gonzalez, J. E., and Stoica, I. 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*, pp. 965–988. USENIX Association, 2024.

Tillet, P., Kung, H. T., and Cox, D. 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, pp. 10–19. Association for Computing Machinery, 2019. ISBN 9781450367196.

Wang, L., Yang, N., Huang, X., Yang, L., Majumder, R., and Wei, F. Large search model: Redefining search stack in the era of llms. *SIGIR Forum*, 57(2):23:1–23:16, 2023.

Xia, H., Zheng, Z., Li, Y., Zhuang, D., Zhou, Z., Qiu, X., Li, Y., Lin, W., and Song, S. L. Flash-llm: Enabling low-cost and highly-efficient large generative model inference with unstructured sparsity. *Proc. VLDB Endow.*, 17(2): 211–224, 2023.

Xiong, H., Bian, J., Li, Y., Li, X., Du, M., Wang, S., Yin, D., and Helal, S. When search engine services meet large language models: Visions and challenges. *CoRR*, abs/2407.00128, 2024.

Ye, L., Tao, Z., Huang, Y., and Li, Y. 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*, pp. 11608–11620. Association for Computational Linguistics, 2024a.

Ye, Z., Lai, R., Lu, B.-R., Lin, C.-Y., Zheng, S., Chen, L., Chen, T., and Ceze, L. Cascade inference: Memory bandwidth efficient shared prefix batch decoding, February 2024b. URL https://flashinfer.ai/2024/02/02/cascade-inference.html.

Yu, G., Jeong, J. S., Kim, G., Kim, S., and Chun, B. 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*, pp. 521–538. USENIX Association, 2022.

Zhao, Z., Fan, W., Li, J., Liu, Y., Mei, X., Wang, Y., Wen, Z., Wang, F., Zhao, X., Tang, J., and Li, Q. Recommender systems in the era of large language models (llms). *IEEE Transactions on Knowledge and Data Engineering*, 36 (11):6889–6907, 2024. doi: 10.1109/TKDE.2024.3392335.

Zheng, L., Yin, L., Xie, Z., Sun, C., Huang, J., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., Barrett, C., and Sheng, Y. Efficiently programming large language models using sglang. 2023.

Zhu, L., Wang, X., Zhang, W., and Lau, R. W. H. 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*, pp. 4945–4957. Association for Computational Linguistics, 2024.