PARALLEL PROMPTING: FAST LLM INFERENCE FOR SHARED-CONTEXT, SHORT-TO-MODERATE OUTPUT

Anonymous authors
Paper under double-blind review

ABSTRACT

We introduce Parallel Prompting, a novel method for efficiently decoding multiple queries that share a common prefix in large language models (LLMs). This scenario occurs naturally in tasks such as document question answering, few-shot learning, and chatbot systems, where many prompts have substantial overlap. Our approach overcomes shortcomings of prior methods, which either leads to the degraded output quality or inefficient cache management. Crucially, we identify that maximizing inference throughput requires a careful balance between attention parallelism and batch size. The theoretical maximum throughput lies at a point determined by the hardware and model specifics, and cannot be achieved by solely increasing batch size or attention parallelism. In contrast to related methods that forbid hybrid batching or require pre-allocated memory for the entire generation, our approach supports flexible batching across multiple sharing groups and enables dynamic, on-demand memory usage. By decoding all queries in parallel with efficient matrix-matrix operations, our method significantly improves throughput and memory utilization without compromising result quality. Experimental results demonstrate that our method can improve end-to-end Llama3-8B latency by up to 4× against competitive baselines on popular datasets, without compromising output quality or accuracy.

1 INTRODUCTION

Batch text generation is a standard paradigm for large language models (LLMs) inference. In many practical scenarios, prompts within a batch often share a common prefix. This setting is prevalent in wide range of use-cases, such as document question answering, few-shot learning, multi-user chat, LLM-as-judges for model evaluation, and LLM-based verification for fact-checking. For instance, chatbots frequently serve diverse users using a shared system prompt, assistant models leverage few-shot exemplars for domain-specific tasks, and programming systems generate multiple candidate solutions to a single problem. As deployment of transformer-based LLMs continues to scale, harnessing these shared prefixes for efficiency becomes increasingly valuable.

Shared prefixes result in overlapping attention key and value representations across sequences, opening the door to specialized optimization techniques. With growing demand comes the necessity for LLMs to efficiently handle prompts containing more shared content, many recent works focus on optimizing LLM inference in this scenario, such as RelayAttention (Zhu et al., 2024) and Hydragen (Juravsky et al., 2024). However, shortcomings of these prior methods hinder their widespread adoption. Hydragen requires pre-allocated memory for the entire generation and RelayAttention assumes that all requests share the same system prompt, a hybrid batch with multiple sharing groups is not supported. Other works such as PromptCache (Gim et al., 2024), vLLM (Kwon et al., 2023) and SGLang (Zheng et al., 2024) optimize the inference by caching the past queries without utilizing the benefit of efficiency of matrix multiplications. SeqBatch prompting, which merges with many queries sequentially within a single prompt, often causes degraded performance (Cheng et al., 2023; Lin et al., 2023).

In this paper, we introduce Parallel Prompting, a novel and simple method for high-throughput, quality-preserving decoding of multiple LLM prompts that share a common prefix. Our approach benefits from increased parallelization during decoding, which is easy to implement efficiently using matrix multiplications in modern GPUs.

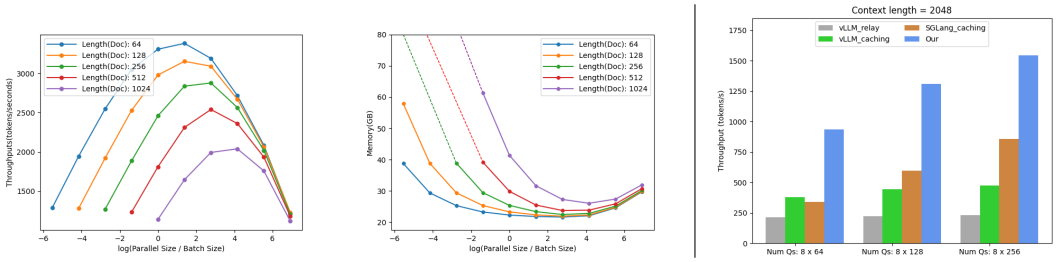

Figure 1: **Left:** Throughputs comparison for a total 1024 questions with different document settings. **Middle:** Memory usage comparison for a total 1024 questions with different document settings. **Right:** Throughputs of different inference methods with CodeLlama-7b-Instruct model on A100 GPU when a large number of questions are queried over long prefixes. Batch size and parallel size need to be balanced to reach the maximum throughput.

To summarize, our work makes the following contributions:

- We propose a simple and effective method leveraging parallel prompting in LLM that allows efficient batching of multiple LLM prompts which share a prefix.
- We conduct extensive experiments and show that our method can achieve improvements in throughput and computational resource management over prior methods across a range of workloads, although there are some workloads for which our proposed method is less efficient than some prior methods.
- We show theoretically and experimentally that maximizing inference throughput for parallel prompting requires a careful balance between attention parallelism and batch size.

Our approach improves throughput and memory utilization without compromising accuracy, with the largest gains when prefix overlap is high and outputs are short to moderate; as unique suffixes lengthen, the relative advantage decreases. While our approach has strong benefits in certain key use-cases, we caution that our proposed method is not superior in all use-cases. For very long generations where query-specific suffixes dominate computation, prompt-agnostic schedulers (e.g., vLLM's default) are preferable. Parallel Prompting complements existing batching strategies and broadens the practical deployment of LLMs in high-throughput evaluation and verification workflows.

2 Background: Attention Mechanism

A core component of the Transformer is the attention computation. Given the sequence of queries $Q \in \mathbb{R}^{N_q \times d}$, keys $K \in \mathbb{R}^{N_{kv} \times d}$, values $V \in \mathbb{R}^{N_{kv} \times d}$, the transformer model computes the attention output $O \in \mathbb{R}^{N_q \times d}$ with the causal masking M as follows:

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

At the start of the generation process, a prefill stage processes the initial sequence of tokens that the LLM will complete. During this stage, the entire prompt is encoded in parallel using a single transformer forward pass. This results in a high number of queries and key-value pairs ($N_q = N_{kv} \gg 1$), making the matrix multiplications in Equation 1 more hardware-friendly.

As the generation continues, completion tokens are decoded sequentially, with each decoding step producing a new token and requiring a forward pass. To speed up this process, a KV cache is used to store the attention keys and values of all previous tokens, eliminating the need to reprocess the entire sequence during each decoding step. Instead, only the most recent token is passed through the model. However, this approach results in a different attention computation where the number of queries is 1 while the number of key-value pairs is still high ($N_q = 1$ and $N_{kv} \gg 1$). This leads to matrix-vector products for the multiplications with K^T and V, making the attention during decoding memory-bound and not utilizing tensor cores.

3 METHOD

Suppose we have a context C and N sentence queries $q_1, \ldots, q_n$ for the context. Let the generation function of original model be $f()$, and suppose the current batch of data with batch size N is $Q = \{q_1, q_2, ..., q_n\}$, the answers to each data are $A = \{a_1, a_2, ..., a_n\}$.

In the situation of standard batch prompting multiple questions Q based on the same context C from the auto-regressive language model, the final answer a_i for q_i as $f(C, q_i)$. In order to improve the inference efficiency, the SeqBatch prompting merges all questions together and generates answer a_i for each q_i as $f(C, Q, a_{1:i-1})$.

However, the answer A_n to the data Q_n is not only conditioned on the task specification but also on $\{a_1, a_2, ...a_{n-1}\}$, which can be viewed as the context of a_n. Therefore, all of the generated answers have a unique effect for the following ones in the batch prompting method, which we refer to as the prompt interference problem.

To tackle this problem, we prefill all questions with Prompt-wise Independent Encoding. Our approach differs from SeqBatch prompting in that it utilizes independent masking initialization during the prefilling stage, drawing on ideas from prepack perfilling (Zhao et al., 2024). The constructed mask matrix for each answer makes sure that it only pays attention to its corresponding question and the shared context. With the specialized attention mask, we are able to compute attention over the shared context and corresponding question as a standalone operation for every answer. While this specialized attention mask does not improve efficiency on its own (in fact, it introduces additional work to initialize a mask for each answer), it allows computing cross-attention much more efficiently over a batch of sequences in the following generation stage.

3.1 PARALLEL GENERATION WITH PROMPT-WISE INDEPENDENT ENCODING

In the prefill stage of our method , the model encodes the prompt in parallel within a single forward pass. During this phase, the LLM takes a prepacked prompt sequence with a modified masking and position encoding to extract the corresponding KV-cache values. Each question's position index follows the end token index of context, which ensures the correct position embedding passing into the model. If the attention status of context is already precached, the prefill process can also be done by providing the context attention status as past kv-cache. We provide the pseudo-codes for our parallel generation process in Algorithm 1.

Since all questions are independent and share a common context, we are able to generate the probability distribution of answers simultaneously. To achieve this, we need to allow the model to generate N tokens at once in each forward pass of the generation stage, which means increasing the number of query vectors in the attention computation by making Q a matrix of dimension $N \times d$.

During the decoding phase, our method generates tokens for different questions simultaneously. In the process of parallel generation, each forward pass would generate N new tokens which is also the number of questions. Since the position of each generated token should be followed by the provided prefix tokens, we have to record the position of all last input tokens and add 1 for them. Also, in order to seamlessly generate the full answers to the provided questions, we update the generated tokens to their corresponding positions in the inputs prompt. Since all the mask attention structures are already defined from the prefill stage, the model only needs to update them with the same pattern in the generation stage.

3.2 THEORETICAL ANALYSIS

In this section, we present a theoretical analysis of the parallel prompting method, focusing on its efficiency gains in LLM inference. We begin by discussing the implications of Amdahl's Law in the context of parallel algorithms, followed by an examination of the speedup and throughput improvements achieved through our approach.

Amdahl's Law provides a theoretical framework for understanding the potential speedup of a task when a portion of it is parallelized. It is defined as:

$$S(N) = \frac{1}{(1-p) + \frac{p}{N}} \tag{2}$$

Algorithm 1 Parallel_Batch_Prompting: Parallel Prompt Generation with Pre-Filling Cache

Require: Shared prefix Doc, unique suffixes Q_{all}, batch size N, parallel size P, language model π_{LLM}
Ensure: List of generated answers by π_{LLM}

Optional: *cache* $\leftarrow$ PRECOMPUTE(π_{LLM}, Doc) ▷ Pre-fill key/value cache with shared prefix
$i \leftarrow 0$
$N_p \leftarrow N/P$ ▷ Number of samples per parallel group
while $i < |Q_{\text{all}}|$ **do**
  $Q_n \leftarrow Q_{\text{all}}[i : i + N]$ ▷ Select mini-batch
  $Q_{np} \leftarrow$ PARALLIZEINTERLEAVE(Q_n, P)
  $prompts \leftarrow$ PREPAREINPUT(Doc, Q_{np}, N_p)
  $masks \leftarrow$ PREPAREMASK($prompts$) ▷ Combine 3D mask and padding
  $answers, output\_mask \leftarrow$ PARALLELGENERATE($\pi_{\text{LLM}}, prompts, masks, P,$ *cache*)
  for $n = 1$ **to** N_p **do**
    for $p = 1$ **to** P **do**
      $final\_answer$.append(DECODE($answers[n, p], output\_mask[n, p]$))
    end for
  end for
  $i \leftarrow i + N$
end while
return $final\_answer$

function PARALLELGENERATE(π_{LLM}, prompts, masks, P, *cache*)
  ▷ Generate outputs in parallel using P groups, leveraging prefilled cache
  Initialize $finished \leftarrow$ False
  Initialize $input\_ids \leftarrow$ TOKENIZE($prompts$)
  while not $finished$ **do**
    $outputs \leftarrow \pi_{\text{LLM}}$.FORWARD($input\_ids, masks,$ *cache*)
    $logits \leftarrow outputs[:, -P:]$
    $next\_tokens \leftarrow$ SAMPLE($logits$)
    $input\_ids \leftarrow$ CONCAT($input\_ids, next\_tokens$)
    if STOPPINGCRITERIA($input\_ids$) **then**
      $finished \leftarrow$ True
    else
      $masks \leftarrow$ UPDATEPARALLELMASK($input\_ids, P$)
    end if
  end while
  return $input\_ids$, $masks$
end function

function PRECOMPUTE(π_{LLM}, Doc)
  ▷ Efficiently pre-fills the LM cache for the shared prefix Doc
  $kv\_cache \leftarrow \pi_{\text{LLM}}$.FORWARD($Doc$)
  return kv_cache
end function

where $S(N)$ is the speedup with N processors, p is the fraction of the task that can be parallelized, $1 - p$ is the fraction that remains serial. This law highlights that the overall speedup is limited by the serial portion of the task. As N increases, the speedup approaches $\frac{1}{1-p}$, indicating diminishing returns if p is not close to 1.

In the context of LLM inference, traditional methods process each query sequentially, leading to inefficiencies due to the serial nature of prompt processing. Our proposed method introduces parallel prompting, allowing multiple queries to be processed simultaneously. This approach effectively maximizes throughput and reduces the time of the LLM's inference task. We measure throughput as queries (prompts) processed (a full output completion is generated) per unit time.

Theorem 1 (Amdahl's Law for Inference Throughput Improvement)**.** *The throughput improvement Δ (tasks processed per unit time above baseline) from using N-way parallel inference is:*

$$\Delta = \frac{N \cdot S(N) - 1}{T_{\text{seq}}} \tag{3}$$

See proofs and further details in Equation A.1.

Proposition 2. *Consider inference on N independent queries using (a) standard batch processing and (b) parallel prompting (packing all queries as independent subsequences in a single sequence with attention masking).*

Let $T_{batch} = T_{setup} + N \cdot T_{\text{MV}}$ be the wall-time for a batch (with matrix-vector attention), and $T_{parallel} = T_{setup} + T_{\text{MM}}$ for parallel prompting (with matrix-matrix attention). Then, the respective throughput values are:

$$\textit{Throughput}_{batch} = \frac{N}{T_{batch}}, \qquad \textit{Throughput}_{parallel} = \frac{N}{T_{parallel}} \tag{4}$$

and

$$\frac{\textit{Throughput}_{parallel}}{\textit{Throughput}_{batch}} = \frac{T_{batch}}{T_{parallel}} = \frac{T_{setup} + NT_{\text{MV}}}{T_{setup} + T_{\text{MM}}} \tag{5}$$

where T_{MV} is per-query wall-time for the matrix-vector attentions, and T_{MM} is wall-time for the matrix-matrix product in the attention.

In practical settings, due to the efficiency of matrix multiplications on a GPU, $T_{\text{MM}} \approx T_{\text{MV}}$. If $T_{setup} \ll T_{\text{MM}}$, then $\textit{Throughput}_{parallel}$ is up to $N\times$ that of standard batching.

While the theoretical analysis suggests significant improvements, practical factors such as communication overhead, memory bandwidth constraints, and synchronization costs can impact actual performance. It is essential to consider these factors when implementing parallel prompting to ensure that the theoretical gains translate into real-world efficiency.

3.3 Throughput Maximization by Balancing Attention Parallelism and Batch Size

The use of batching is a crucial technique to enhance throughput in LLM inference. Through batched decoding, each forward pass of the model processes the latest token from multiple sequences concurrently rather than just one. This approach amplifies the arithmetic intensity of transformer components, such as the multilayer perceptron (MLP) blocks, and facilitates the use of hardware-friendly matrix multiplications.

However, the computation intensity of attention does not inherently benefit from batching, as each sequence possesses its distinct key and value matrix. Consequently, while other model components can leverage tensor cores during batched decoding, attention is required to be computed using numerous independent matrix-vector products. Our parallel generation technique aims to address this by enhancing the computation intensity of attention.

Proposition 3 (Throughput Maximization)**.** *Let P be the parallel size (number of independent queries packed into a sequence for matrix-matrix attention), B the batch size (number of such sequences processed in parallel), and $P \cdot B \leq S^*$ a hardware resource constraint (e.g., total token capacity).*

Let $T_{\text{attn}}(P)$ denote the attention computation cost (function of P), and $T_{\text{mlp}}(B)$ denote the MLP/other backend (function of B).

Then, the throughput (queries per unit time) satisfies:

$$\text{Throughput}(P, B) = \frac{P \cdot B}{T_{\text{attn}}(P) + T_{\text{mlp}}(B)} \tag{6}$$

and maximal throughput is achieved at

$$(P^*, B^*) = \arg\max_{P \cdot B \leq S^*} \frac{P \cdot B}{T_{\text{attn}}(P) + T_{\text{mlp}}(B)} \tag{7}$$

where $T_{\text{attn}}(P)$ generally improves with P up to a hardware limit (then degrades), and $T_{\text{mlp}}(B)$ improves with B up to a limit.

The maximizing pair (P^*, B^*) is found by balancing optimal matrix-matrix utilization for attention and optimal batch size for MLP efficiency. The throughput function is quasi-concave in (P, B) under natural hardware scaling assumptions for transformer kernels. The theoretical maximum exists at an interior point determined by hardware and model specifics, and is not achieved by maximizing either P or B alone.

Comparing our method to prior work, RelayAttention (Zhu et al., 2024) does not support batching with different prefixes, as it necessitates a more complex implementation of fused operators in CUDA for hybrid batching with multiple sharing groups. Hydragen (Juravsky et al., 2024) requires a batched document with the same number of questions, and its implementation also has a constraint on the question (prompt) length.

Our method integrates seamlessly with the batching technique. By batching texts with multiple unique documents and corresponding questions, efficiency can be improved further. Parallel generation with batching provides two distinct advantages: firstly, inference throughput is further amplified by batching with multiple unique prefix documents; secondly, it enables the balancing of batch size and sequence length for model input, optimizing overall performance.

4 Experiments

To enable controlled, fine-grained analysis of scaling behavior and performance factors, we conduct systematic scaling experiments on smaller models—CodeLlama-7b-Instruct (Rozière et al., 2024), Sheared-LLaMA-1.3B (Xia et al., 2024), and LLaMa-160m (Miao et al., 2023)—using synthetic datasets. This setup balances computational feasibility with systematic study and demonstrates the reliability and effectiveness of our approach.

Next, we evaluate on downstream tasks with the Llama 3-8B model on reading comprehension datasets as a case study to demonstrate the benefits of our method on an example task. Across all models, we employ a consistent parallel generation method to predict the next set of multiple-answer tokens; all experiments are run on a single NVIDIA A100-80GB GPU, with implementations in PyTorch using the HuggingFace architecture (Wolf et al., 2020). Additional details are provided in Appendix B.

4.1 Scaling Experiments

For the experiments in this section, we constructed synthetic data following Juravsky et al. (2024) with different lengths and numbers of unique documents and various numbers of questions. The content of the constructed document is a subset of War and Peace (Tolstoy, 1869), modified to include procedurally constructed sentences.

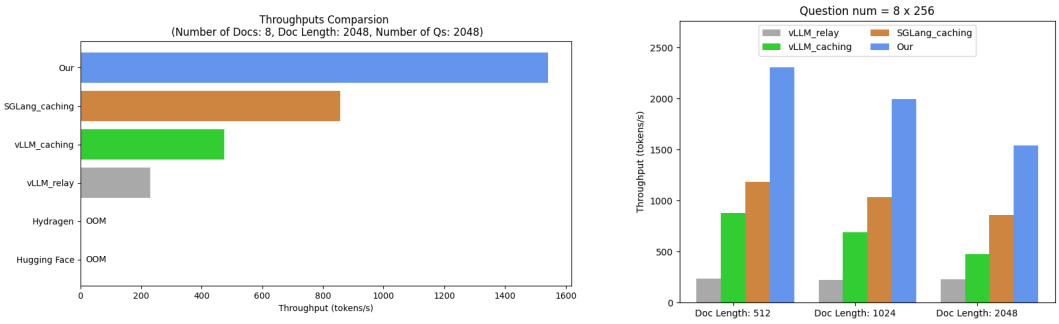

Figure 2: **Left:** Memory usage of HuggingFace and Hydragen could cause memory issues when a large number of questions are being queried. **Right:** Throughputs of different methods when the number of unique documents changes in the LLM inference.CodeLlama-7b-Instruct attention inference Throughput w.r.t. different lengths of shared documents (A100-SXM4-80GB GPU). We set the number of queries to 256, the lengths for each context sweeps over the list of [512, 1024, 2048], the length of each query to 12, the length of generated tokens to 5.

Memory and Throughput Scaling. To illustrate the memory and throughput scaling behavior of our method, we split the results by output length for clarity. Figure 3 shows memory usage and throughput.

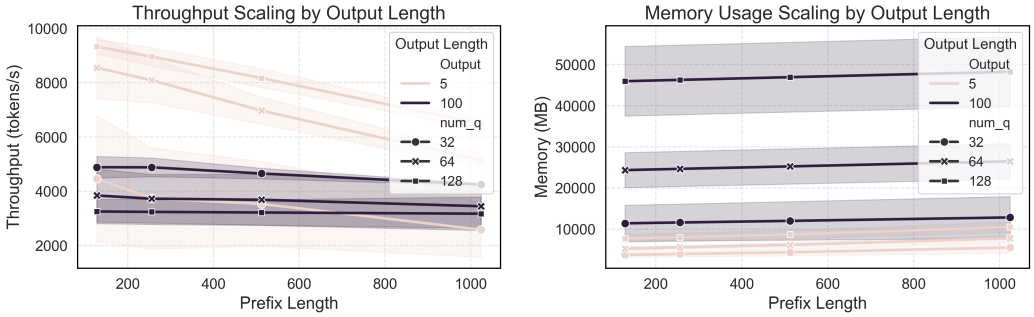

Figure 3: **Left:** Throughput (tokens generated per second) as a function of prefix length and output length. Each line corresponds to a specific output length, with markers denoting the number of questions processed in parallel. The shaded band represents the 95% confidence interval. The plot demonstrates that throughput generally decreases as prefix length and output length increase, due to greater computational load. The comparison across settings provides insight into the efficiency and stability of the method under different workload sizes and output requirements. **Right:** We illustrate how GPU memory usage varies with prefix length and output length across different experimental settings. The plot shows that memory usage increases with both longer prefixes and higher output lengths, highlighting the scalability of the method under varying batch and output configurations. Memory usage is reported for 4 documents and 32 questions per batch.

We construct synthetic data with various document lengths and various numbers of questions to evaluate our method. We compare the throughput of prefix sharing methods like SGlang with caching, vLLM with caching, vLLM with RelayAttention, and our parallel prompting on the generated synthetic data. As shown in Figure 1, as the number of questions increases, our parallel generation method continues to outperform other methods without the decrease in generation quality. Full results for all settings with memory usages is provided in Tables 3 and 5 in the Appendix.

Performance on Longer Outputs. Evaluating only with very short output lengths is not representative of many real-world workloads (e.g., chain-of-thought, code generation, summarization). To address this, we conduct experiments varying output length up to 300 tokens. Results on our syntactic dataset show that Parallel Prompting consistently delivers throughput gains over the vLLM method

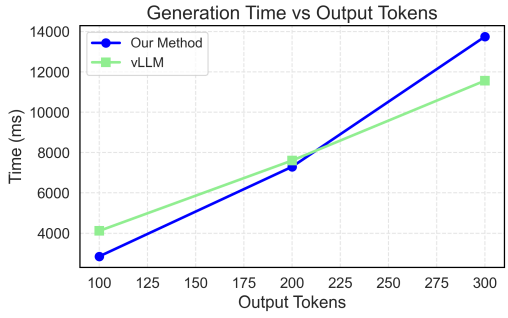

Figure 4: Comparison of generation time versus output tokens for our method and vLLM. As the number of output tokens increases, both methods require more time; however, our method consistently achieves lower generation time for shorter outputs and remains competitive as the output length grows. The blue line represents our method, while the light green line represents vLLM, both evaluated with 4 documents and 32 questions per batch.

up to approximately 200 output tokens per question. As an example, for four unique documents with 4×32 questions, our method required 7,295 milliseconds (throughput $\approx$ 3,500 tokens/sec), while the vLLM method takes 7,605 milliseconds (throughput $\approx$ 3,360 tokens/sec). When the output length exceeds 200 tokens, vLLM may offer a greater advantage.

Number of Questions We run our benchmarks on CodeLlama-7b-Instruct (Rozière et al., 2024) with one A100-80GB GPU with various numbers of questions and documents in Figure 1. We compared throughputs of different methods when the number of queries changes in the LLM inference. CodeLlama-7b-Instruct attention inference Throughput w.r.t. number of queries (A100-SXM4-80GB GPU). We set the length of context to 2048, the length of each query to 12, and the length of generated tokens to 5. In Table 2, we fix the document length to 512 tokens and sweep over the question size from a range and generating five tokens per question. As the number of queries increases, the throughput keeps improving on both 1B and 7B models. When the number of queries is small, non-attention operations contribute significantly to decoding time. At these small batch sizes, some methods spend more time staging document KV cache. As the number of questions grows at a certain level, attention over the prefix becomes increasingly expensive, and our parallel generation saves more time for attention computation.

Batch Size vs. Parallel Size We run experiments querying a range of fixed prompts with different batch sizes. Interestingly, maximizing the parallel size(minimizing the batch size) does not always ideal. In Figure 1, the best throughput performance is reached when a certain balance is hold between the parallel size and the batch size. This situation also happens for models with various sizes (7B (Grattafiori et al., 2024),1B (Rozière et al., 2024)). In Table 2, we also observe that the best throughput performance is reached by balancing the parallel size and the batch size. The best number of parallel sizes balances the cost of computation in the arithmetic intensity of the transformer components such as the multilayer perceptron (MLP) blocks and intensity of attention.

Memory Scaling Experiments To systematically study memory and throughput scaling, we conducted experiments varying shared prefix length (128, 256, 512, 1024 tokens), output length (5 vs 100 tokens), number of unique prefixes (num_doc: 4 vs 8), and number of questions per prefix (num_q: 32, 64, 128). Our results reveal several key patterns: (1) Output length is the dominant driver of memory usage, followed by num_doc and context length, with num_q having a smaller but non-negligible effect. (2) Long outputs dominate memory via KV cache growth across all decode steps. (3) num_doc has a much larger impact when output is long, as a longer context is carried through every generated token. (4) Longer shared prefixes add memory, but the effect is modest compared to output length and num_doc, consistent with effective prefix sharing across the batch.

Evaluating the memory usage during the inference process is critical to understanding the trade-offs of our method fully, especially when we increase the context length. With our method, memory usage could become less as it usually requires a smaller batch size and does not increase the memory used by the KV-Cache compared to the Standard baseline. We conduct the experiment with different settings for querying 1024 questions with different document lengths in Figure 1. Our method and the standard baseline both utilize the Flash Attention algorithm as the backend implementation. A full comparison of memory usage with baseline method is in Table 6 in the Appendix. In Figure 2, some methods suffer the out-of-memory issue when a large number of questions are queried.

4.2 Case Study: Performance on Question Answering

We perform a case study to demonstrate the benefits of our method. We focus on reading comprehension as an example task, as it is a well-studied task. We evaluate on three question answering (QA) datasets: SQUAD (Rajpurkar et al., 2016), QuAC (Choi et al., 2018), and DROP (Dua et al., 2019) with Llama 3-8b (Grattafiori et al., 2024).

The Table 1 compares the generation time of standard, SeqBatch prompting, Hydragen, SGLang, vLLM, vLLM with relay attention and our parallel prompting methods. The result shows that parallel prompting performs consistently better than standard and batch prompting on the latency of generation while remains the same quality of outputs as the standard prompting over all datasets. We use the current latest version, 0.6.4, of the vLLM package, which uses the PagedAttention algorithm. vLLM avoids redundant storage of the prefix, allowing much larger batch sizes to be tested. Additionally,

Table 1: Comparison of generation time and performance for downstream tasks with different methods on average of five times with Llama 3 8B model on A100-80G. Std denotes the across-run standard deviation of the time. F1 is computed as the harmonic mean of precision and recall in extractive QA.

Method	SQuAD			QuAC			DROP		
	Times(s)	Std	F1(%)	Time(s)	Std	F1(%)	Time(s)	Std	F1(%)
Standard	1277	0.08	87.2	3512	0.06	34.0	1330	0.08	58.1
SeqBatch	566	0.21	84.2	386	0.10	29.1	1007	0.41	42.5
Hydragen	1651	20.9	87.1	1230	6.74	34.0	471	3.85	58.2
SGLang	337	0.49	87.4	854	0.17	32.7	377	0.56	58.5
vLLM	369	0.46	87.4	889	0.57	32.8	413	0.44	58.5
vLLM-RA	365	0.21	87.3	469	0.15	32.8	179	0.51	58.5
Parallel	167	0.16	87.2	243	0.32	33.9	110	0.09	58.1

because of this non-redundant storage, PagedAttention can achieve a higher GPU cache hit rate when reading the prefix, reducing the cost of redundant reads.

5 Related work

Recent advancements in language modeling have delved into the prediction of multiple tokens simultaneously to enhance both efficiency and performance. Notable works such as (Miao et al., 2024; Leviathan et al., 2023; Wu et al., 2024) focus on speculative decoding methods, where potential future sequences are built and verified to expedite inference. Similarly, (Gloeckle et al., 2024) and (Cai et al., 2024) propose predicting multiple future tokens using different output heads, thereby speeding up the inference process. Efforts to increase throughput in LLM inference have led to various innovative techniques aimed at optimizing GPU utilization and improving throughput. (Dao et al., 2022) and (Sheng et al., 2023) aim to improve memory usage efficiency, enabling higher throughput in generative inference tasks. (Jin et al., 2023) schedules prompts based on estimated output sequence lengths to optimize GPU usage. (Gim et al., 2024) proposes reusing precomputed caches in a predefined schema to reduce latency. (Sun et al., 2024) applies dynamic sparse KV caching in decoding to accelerate long sequence generation. Efficient prompting techniques could also increase the throughput of LLM.(Cheng et al., 2023) groups multiple questions in a single prompt, though it will lead to performance degradation when the number of questions increases. (Zhao et al., 2024) enhances throughput during the prefilling stage by prepacking data. (Ning et al., 2024) uses the skeleton of the answer to batch-generate the final answer. To avoid the KV cache duplication, existing work (Kwon et al., 2023) vLLM uses its PagedAttention and paged memory management to point multiple identical input prompts to only one physical block across multiple queries. Also, (Juravsky et al., 2024) proposes a decomposition of attention computation of shared prefixes and unique suffixes. (Lu et al., 2024) increases efficiency by sharing cache in the encoder-decoder model for decomposable tasks. Compared with the above methods, our work introduces a novel inference technique that allows LLMs to leverage GPU parallel capacity to improve inference throughput and memory utilization without degrading reasoning performance.

6 Conclusion

We introduce an efficient parallel prompting method for decoding prompt queries in parallel. We conduct experiments with multiple down stream datasets, generate synthetic data, and show our method achieves improvements in throughput and computational resource management, offering a robust solution for different tasks in LLMs.

Limitations

Skewed Generation Lengths Our method achieves the highest throughput gains when suffix lengths are similar, and performance may degrade when generation lengths are highly skewed during decoding. To mitigate this, we propose several practical strategies: In cases where generation lengths

become highly unbalanced, the system can fall back to standard inference. In real-world applications, expected output length can often be heuristically estimated based on properties such as question and context length. This enables grouping questions with similar expected output lengths, minimizing skew. More advanced solutions, such as dynamic batching (e.g., as introduced in Verl), could be adopted to support streaming scenarios and further optimize batching efficiency.

Prompt-Agnostic Batching Our method's gains are largest when there is a clear shared-prefix structure and output lengths are short to moderate. As the length of unique suffixes increases, the benefit of parallel generation diminishes, since more computation must be performed individually for each query. For very long outputs, prompt-agnostic batching (such as vLLM's default scheduling) may outperform our approach. We recommend a hybrid scheduling policy in production, using Parallel Prompting for workloads with substantial shared context and prompt-agnostic batching for others. This method is designed to complement, not replace, existing batching strategies.

Reproducibility Statement

We have taken several steps to facilitate reproducibility. Assumptions and proofs for all theoretical claims are provided in Appendix [A], which states all conditions under which the results hold. Experimental settings—including datasets, preprocessing, model configurations, training schedules, hyperparameters, and evaluation protocols in Section [X] (Experiments). An anonymized, self-contained supplementary .zip archive includes source code and scripts to reproduce the main tables/figures and ablations. Known limitations, potential failure modes, and scope of applicability are discussed in Section Limitations. Any deviations from the default procedures or additional implementation notes are included in Appendix [B].

References

Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D Lee, Deming Chen, and Tri Dao. Medusa: Simple llm inference acceleration framework with multiple decoding heads. *arXiv preprint arXiv:2401.10774*, 2024.

Zhoujun Cheng, Jungo Kasai, and Tao Yu. Batch prompting: Efficient inference with large language model apis. *arXiv preprint arXiv:2301.08721*, 2023.

Eunsol Choi, He He, Mohit Iyyer, Mark Yatskar, Wen-tau Yih, Yejin Choi, Percy Liang, and Luke Zettlemoyer. Quac: Question answering in context. *arXiv preprint arXiv:1808.07036*, 2018.

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

Dheeru Dua, Yizhong Wang, Pradeep Dasigi, Gabriel Stanovsky, Sameer Singh, and Matt Gardner. Drop: A reading comprehension benchmark requiring discrete reasoning over paragraphs. *arXiv preprint arXiv:1903.00161*, 2019.

In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt cache: Modular attention reuse for low-latency inference. *Proceedings of Machine Learning and Systems*, 6:325–338, 2024.

Fabian Gloeckle, Badr Youbi Idrissi, Baptiste Rozière, David Lopez-Paz, and Gabriel Synnaeve. Better & faster large language models via multi-token prediction. *arXiv preprint arXiv:2404.19737*, 2024.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle

Pintz, Danny Livshits, Danny Wyatt, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Francisco Guzmán, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Govind Thattai, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jack Zhang, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Karthik Prasad, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Kushal Lakhotia, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Maria Tsimpoukelli, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Ning Zhang, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohan Maheswari, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vítor Albiero, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaofang Wang, Xiaoqing Ellen Tan, Xide Xia, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aayushi Srivastava, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Amos Teo, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Dong, Annie Franco, Anuj Goyal, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Ce Liu, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Cynthia Gao, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Eric-Tuan Le, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Filippos Kokkinos, Firat Ozgenel, Francesco Caggioni, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hakan Inan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Hongyuan Zhan, Ibrahim Damlaj, Igor Molybog, Igor Tufanov, Ilias Leontiadis, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Janice Lam, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan Saxena, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kiran Jagadeesh, Kun Huang, Kunal Chawla, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish

Bhatt, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Miao Liu, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikhil Mehta, Nikolay Pavlovich Laptev, Ning Dong, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Rangaprabhu Parthasarathy, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Russ Howes, Ruty Rinott, Sachin Mehta, Sachin Siby, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Mahajan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shishir Patil, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Summer Deng, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Koehler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaojian Wu, Xiaolan Wang, Xilun Wu, Xinbo Gao, Yaniv Kleinman, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yu Zhao, Yuchen Hao, Yundi Qian, Yunlu Li, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, Zhiwei Zhao, and Zhiyu Ma. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

Yunho Jin, Chun-Feng Wu, David Brooks, and Gu-Yeon Wei. s^3: Increasing gpu utilization during generative inference for higher throughput. *Advances in Neural Information Processing Systems*, 36:18015–18027, 2023.

Jordan Juravsky, Bradley Brown, Ryan Ehrlich, Daniel Y Fu, Christopher Ré, and Azalia Mirhoseini. Hydragen: High-throughput llm inference with shared prefixes. *arXiv preprint arXiv:2402.05099*, 2024.

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

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

Jianzhe Lin, Maurice Diesendruck, Liang Du, and Robin Abraham. Batchprompt: Accomplish more with less. *arXiv preprint arXiv:2309.00384*, 2023.

Bo-Ru Lu, Nikita Haduong, Chien-Yu Lin, Hao Cheng, Noah A Smith, and Mari Ostendorf. Encode once and decode in parallel: Efficient transformer decoding. *arXiv preprint arXiv:2403.13112*, 2024.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Rae Ying Yee Wong, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. Specinfer: Accelerating generative llm serving with speculative inference and token tree verification, 2023.

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

Xuefei Ning, Zinan Lin, Zixuan Zhou, Zifu Wang, Huazhong Yang, and Yu Wang. Skeleton-of-thought: Prompting llms for efficient parallel generation. In *The Twelfth International Conference on Learning Representations*, 2024.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text, 2016. URL https://arxiv.org/abs/1606.05250.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024. URL https://arxiv.org/abs/2308.12950.

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pp. 31094–31116. PMLR, 2023.

Hanshi Sun, Zhuoming Chen, Xinyu Yang, Yuandong Tian, and Beidi Chen. Triforce: Lossless acceleration of long sequence generation with hierarchical speculative decoding. *arXiv preprint arXiv:2404.11912*, 2024.

Leo Tolstoy. *War and Peace*. 1869.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, pp. 38–45, 2020.

Pengfei Wu, Jiahao Liu, Zhuocheng Gong, Qifan Wang, Jinpeng Li, Jingang Wang, Xunliang Cai, and Dongyan Zhao. Parallel decoding via hidden transfer for lossless large language model acceleration. *arXiv preprint arXiv:2404.12022*, 2024.

Mengzhou Xia, Tianyu Gao, Zhiyuan Zeng, and Danqi Chen. Sheared llama: Accelerating language model pre-training via structured pruning, 2024. URL https://arxiv.org/abs/2310.06694.

Siyan Zhao, Daniel Israel, Guy Van den Broeck, and Aditya Grover. Prepacking: A simple method for fast prefilling and increased throughput in large language models. *arXiv preprint arXiv:2404.09529*, 2024.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *arXiv preprint arXiv:2312.07104*, 2024.

Lei Zhu, Xinjiang Wang, Wayne Zhang, and Rynson WH Lau. Relayattention for efficient large language model serving with long system prompts. *arXiv preprint arXiv:2402.14808*, 2024.

A Appendix

A.1 Proof of Theorem 1.

Amdahl's Law for Inference Throughput Improvement The throughput improvement Δ (tasks processed per unit time above baseline) from using N-way parallel inference is:

$$\Delta = \frac{N \cdot S(N) - 1}{T_{\text{seq}}} \tag{8}$$

Assumptions:

- Each inference computation can be split into a parallelizable fraction and a sequential fraction.
- There are N independent queries, each requiring T_{seq} execution time if performed sequentially.
- There is no communication, scheduling, or parallelization overhead. Negligible coordination or resource contention.
- N processors are available, and the parallel workload is divided equally among them. In parallel, independent N queries are processed in time $T_{\text{par}}(N) = T_{\text{seq}}/S(N)$, where $S(N)$ is given by Amdahl's law Equation 2

Proof of Theorem 1. The sequential throughput is $\frac{1}{T_{\text{seq}}}$. With parallel prompting, the time to process N queries is $T_{\text{par}}(N)$, so the parallel throughput is $\frac{N}{T_{\text{par}}(N)}$. The improvement is:

$$\Delta = \frac{N}{T_{\text{par}}(N)} - \frac{1}{T_{\text{seq}}}$$

Assuming $T_{\text{par}}(N) = \frac{T_{\text{seq}}}{S(N)}$, we substitute to get:

$$\Delta = \frac{N}{\frac{T_{\text{seq}}}{S(N)}} - \frac{1}{T_{\text{seq}}} = \frac{N \cdot S(N)}{T_{\text{seq}}} - \frac{1}{T_{\text{seq}}} = \frac{N \cdot S(N) - 1}{T_{\text{seq}}}$$

□

A.2 Assumptions of Proposition 2

Let $T_{\text{batch}} = T_{\text{setup}} + N \cdot T_{\text{MV}}$ be the wall-time for a batch (with matrix-vector attention), and $T_{\text{parallel}} = T_{\text{setup}} + T_{\text{MM}}$ for parallel prompting (with matrix-matrix attention). Then, the respective throughput values are:

$$\text{Throughput}_{\text{batch}} = \frac{N}{T_{\text{batch}}}, \qquad \text{Throughput}_{\text{parallel}} = \frac{N}{T_{\text{parallel}}} \tag{9}$$

and

$$\frac{\text{Throughput}_{\text{parallel}}}{\text{Throughput}_{\text{batch}}} = \frac{T_{\text{batch}}}{T_{\text{parallel}}} = \frac{T_{\text{setup}} + NT_{\text{MV}}}{T_{\text{setup}} + T_{\text{MM}}} \tag{10}$$

where T_{MV} is per-query wall-time for the matrix-vector attentions, and T_{MM} is wall-time for the matrix-matrix product in the attention.

Assumptions:

- The model and hardware support this masking and packing; T_{MV} and T_{MM} are measured compatibly.
- Time for setup is equal for standard batch processing and parallel prompting,
- N is small enough to avoid exceeding hardware or memory limits for both methods.

A.3 Assumptions of Proposition 3

Throughput Maximization Let P be the parallel size (number of independent queries packed into a sequence for matrix-matrix attention), B the batch size (number of such sequences processed in parallel), and $P \cdot B \leq S^*$ a hardware resource constraint (e.g., total token capacity).

Let $T_{\text{attn}}(P)$ denote the attention computation cost (function of P), and $T_{\text{mlp}}(B)$ denote the MLP/other backend (function of B).

Then, the throughput (queries per unit time) satisfies:

$$\text{Throughput}(P, B) = \frac{P \cdot B}{T_{\text{attn}}(P) + T_{\text{mlp}}(B)} \tag{11}$$

and maximal throughput is achieved at

$$(P^*, B^*) = \arg\max_{P \cdot B \leq S^*} \frac{P \cdot B}{T_{\text{attn}}(P) + T_{\text{mlp}}(B)} \tag{12}$$

where $T_{\text{attn}}(P)$ generally improves with P up to a hardware limit (then degrades), and $T_{\text{mlp}}(B)$ improves with B up to a limit.

Assumptions:

- P queries packed per prompt, B prompts in a batch, $PB \leq S^*$ (resource or hardware constraint).
- Model/hardware supports this arrangement; $T_{\text{attn}}(P)$ and $T_{\text{mlp}}(B)$ are the attention/MLP module wall times.
- $T_{\text{attn}}(P)$, $T_{\text{mlp}}(B)$ are nonincreasing (improve) up to hardware limits, then nonmonotone.

B Technical Appendices and Supplementary Material

The decision to use different models and datasets for the analytical and ablation studies, as compared to the main downstream task evaluations, is motivated by both practical and scientific considerations. Large models like Llama 3-8B are computationally intensive, making it challenging to run extensive ablation and scaling experiments across a wide range of parameters. By using smaller models and synthetic datasets for these studies, we are able to systematically vary key factors (such as batch size, prefix length, and number of queries) and isolate the effects of our method in a controlled environment. This approach enables us to provide deeper insights into the scaling laws, bottlenecks, and generalization of our method, while reserving the large-scale, real-world benchmarks for the main results. We believe this combination offers a comprehensive and rigorous evaluation of our approach.

Memory Usage on QuAC The observed increase in memory usage for the Parallel method on QuAC results from dynamically maximizing batch sizes during inference. Our approach allows processing more examples in a fixed memory footprint, improving throughput. To validate this, we reduced the maximum allowed batch size during inference on QuAC and observed a significant drop in memory usage, while still demonstrating substantial speedup over the baseline with the maximum possible batch size. For transparency, Table 4 lists the results across different batch size settings with our method. This demonstrates that our method flexibly trades off memory and throughput by adjusting batch size, and can achieve substantial speedup even at lower memory footprints.

Document Length Now, we run a similar experiment, except now we hold the number question as 256 for each document and sweep the document length among the list $[512, 1024, 2048]$ in Figure 2. Even though the throughput decreases as the prefix grows, our parallel generation method continues to outperform other methods. in Appendix B - for smaller models and more parallel questions, the speedup can

We perform additional experiments over different models, prefix lengths, and batch sizes (see Appendix B). For smaller models and larger numbers of parallel questions, the speedup can be even greater than what is shown in Table 7.

Table 2: Comparing the throughput (tokens/second) using parallel Batching with different Batch sizes of parallel generation on 1B and 7B Llama model when the $doc\_len = 512 \| q\_len = 12 \| ans\_len = 5$.

#queries	Batch Size	Throughput-1B	Throughput-7B
128	1	4283	1931
	2	4625	1843
	4	3654	1468
	8	2850	1018
256	1	5911	2115
	2	6384	2250
	4	5748	2071
	8	4959	1615
512	1	5419	1850
	2	6845	2214
	4	7725	2382
	8	7181	2146

Table 3: Memory Usage (MB) and Throughput (tokens/s) for Output Length 5

Prefix	num_doc	num_q	Memory (MB)	Throughput (tok/s)
128	4	32	3187	2144
128	8	32	4324	6794
128	4	64	4798	7412
128	8	64	5767	9700
128	4	128	6735	9060
128	8	128	8724	9602
256	4	32	3290	1875
256	8	32	4622	5605
256	4	64	5095	7264
256	8	64	6073	8928
256	4	128	7039	8627
256	8	128	9041	9304
512	4	32	3512	1976
512	8	32	5260	5098
512	4	64	5687	6479
512	8	64	6684	7472
512	4	128	7671	7831
512	8	128	9727	8520
1024	4	32	4143	1573
1024	8	32	6906	3601
1024	4	64	7135	4882
1024	8	64	8426	5404
1024	4	128	9261	6282
1024	8	128	11751	6689

Sequence Length vs. Computation Gains Trade-off. Both theory and empirical results confirm that throughput increases with batch/parallel size up to a point—after which the computational overhead of longer input sequences (from packed prompts) outweighs the matrix-matrix compute advantage. For example, on A100s, parallel sizes between 32 and 64 are optimal for typical workloads.

Compatibility with Speculative Decoding. Parallel Prompting (fanning out multiple suffixes at lock-step) is designed for simultaneous multi-query generation, while speculative decoding focuses on verifying a single sequence. These are distinct but potentially complementary: speculative decoding could be performed within each branch created by Parallel Prompting, or adapted to verify multiple shared-prefix continuations in parallel.

Table 4: QuAC: Inference Time and Memory Usage for Different Batch Sizes (Parallel Method)

Batch Size	Inference Time (s)	Memory (GB)
Baseline	1799	55.0
8	872	16.9
16	677	19.8
32	420	24.7
64	352	33.1
128	342	54.0

Developer Overhead and Practical Adoption. In many production stacks, the shared-prefix boundary is already explicit: for example, retrieval-augmented generation (RAG) pipelines concatenate retrieved context (prefix) with a question (suffix), and batched APIs naturally group queries under a common header or instruction. In these settings, enabling Parallel Prompting requires only providing: (1) the token span (or delimiter) for the shared prefix, and (2) a list of per-query suffixes. This makes practical adoption straightforward in most modern LLM serving pipelines.

Table 5: Memory Usage (MB) and Throughput (tokens/s) for Output Length 100

Prefix	num_doc	num_q	Memory (MB)	Throughput (tok/s)
128	4	32	7031	4490
128	8	32	15814	5286
128	4	64	20104	4825
128	8	64	28617	2868
128	4	128	37509	3704
128	8	128	54429	2810
256	4	32	7131	4540
256	8	32	16109	5231
256	4	64	20399	4624
256	8	64	28927	2834
256	4	128	37829	3705
256	8	128	54761	2780
512	4	32	7333	4462
512	8	32	16689	4852
512	4	64	20968	4627
512	8	64	29545	2752
512	4	128	38472	3692
512	8	128	55433	2747
1024	4	32	7766	4289
1024	8	32	17932	4217
1024	4	64	22174	4262
1024	8	64	30787	2639
1024	4	128	39751	3792
1024	8	128	56787	2559

Table 6: Comparison of memory usage with different methods with Llama 3 8B model on A100-80G.

Dataset	Method	Time(s)	Memory(GB)
SQuAD	Standard	590	55.7
	Parallel	168	48.6
QuAC	Standard	1799	55.0
	Parallel	352	33.1
DROP	Standard	654	54.3
	Parallel	111	36.1

Effect of Model Size The performance of LLM's generation can be affected by various factors such as number of queries, batch size and the length of prefixes. We also run experiments with various configurations with CodeLlama-7b-Inst (Rozière et al., 2024) and Sheared-LLaMA-1.3B (Xia et al., 2024) since different model sizes could also affect generation performance. See Table 7 for results.

Table 7: Comparing the throughput using parallel Batching with 7B and 1B Llama model with different lengths of doc length when $q\_len = 12 \| q\_num = 128 \| ans\_len = 5$ and the number of unique doc content equals 8. As the content length increases, the degradation of throughput performance becomes severe.

doc_len	$Throughput(1B)(tokens/second)$	$Throughput(7B)(tokens/second)$
256	9512	2750
512	8199	2430
1024	6591	1924