RelayAttention for Efficient Large Language Model Serving with Long System Prompts

Anonymous ACL submission

Abstract

Practical large language model (LLM) services may involve a long system prompt, which specifies the instructions, examples, and knowledge documents of the task and is reused across numerous requests. However, the long system prompt causes throughput/latency bottlenecks as the cost of generating the next token grows w.r.t. the sequence length. This paper aims to improve the efficiency of LLM services that involve long system prompts. Our key observation is that handling these system prompts requires heavily redundant memory accesses in existing causal attention computation algorithms. Specifically, for batched requests, the cached hidden states (*i.e.*, key-value pairs) of system prompts are transferred from off-chip DRAM to on-chip SRAM multiple times, each corresponding to an individual request. To eliminate such a redundancy, we propose RelayAttention, an attention algorithm that allows reading these hidden states from DRAM exactly once for a batch of input tokens. RelayAttention is a free lunch: it maintains the generation quality while requiring no model retraining, as it is based on a mathematical reformulation of causal attention.

1 Introduction

After around one decade of rapid development (Sutskever et al., 2014; Vaswani et al., 2017; Radford et al., 2018; OpenAI, 2023b), we have experienced a revolution of large language models (LLMs) over the past year. LLMs like GPT-4 (OpenAI, 2023b) and Gemini (Google, 2023b) are so powerful that they can now serve as programming copilots (Chen et al., 2021; GitHub, 2022), universal chatbots (Google, 2023a; OpenAI, 2022), computer assistants (Microsoft, 2023a) and other roles that penetrate our daily life. However, the high inference cost of these large models has become a substantial obstacle to serving more people (Kwon et al., 2023). It is therefore important to improve the hardware utilization so that LLMs can have a higher throughput within a fixed hardware budget.

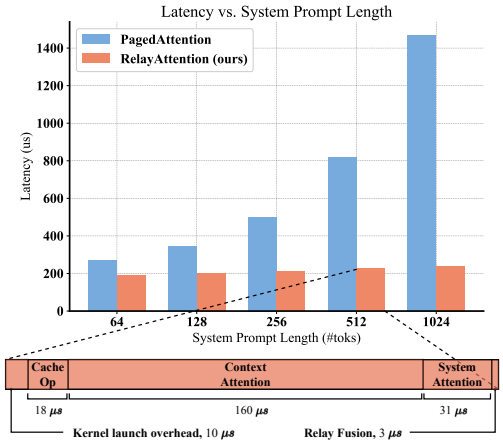


Figure 1: Llama-30B attention inference latency w.r.t. system prompt length (A40 GPU, batch size 32). We set the length of (request-specific) contexts, which include user prompts and previously generated tokens, to 128.

LLM services commonly use an application-specific system prompt (OpenAI, 2023a) to specify the task's instructions. The system prompt is concatenated with the user prompt as the full input to the LLM for response generation and is shared by all requests to a service. The system prompt becomes long if the service provider wants to provide detailed guidelines and examples for better response quality or apply more restrictions/policies for ethical safety. As the sequence length that LLMs can process grows (Anthropic, 2023; Chen et al., 2023b; DeepSeek-AI et al., 2024), some emerging professional applications, such as legal analysis (Cui et al., 2023; Nay et al., 2023), healthcare applications (Steinberg et al., 2021; Rasmy et al., 2021), and the shopping assistant example shown in Fig. 2, may include one or more knowledge documents to provide domain-specific knowledge, resulting in even longer system prompts. Although long system prompts are beneficial to improving the generation quality or enabling new applications, they also pose a challenge to the LLM

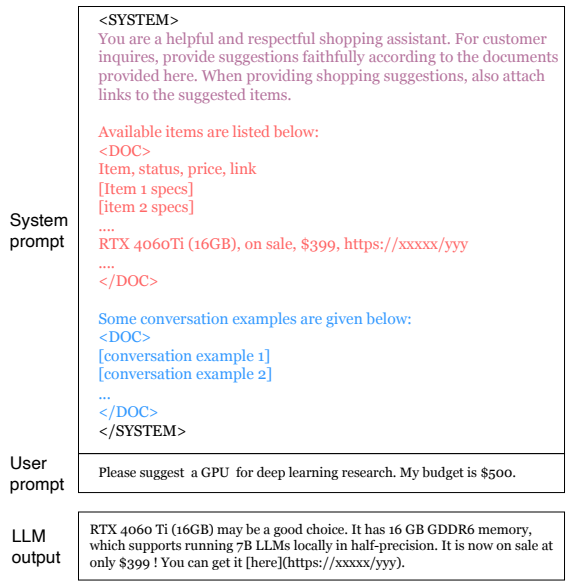


Figure 2: A system prompt may include instructions, knowledge documents and few-shot examples. Here, we use the shopping assistant as an example application.

service: the inference throughput and latency of the service can be heavily degraded, thus increasing the per-request cost. This is inherently caused by the causal attention, in which each new token is generated by "looking at" *all precedent* ones.

In this paper, we propose a novel approach to mitigate the efficiency problem of using long system prompts in LLM services. Our key observation is that there are not only redundant *memory footprint* (Kwon et al., 2023) and computations (Gim et al., 2023) corresponding to the system prompt, but also unnecessary *memory accesses* during causal attention computation. Specifically, while the system prompt is shared by all requests, its hidden states (*i.e.*, key-value pairs) are read from DRAM multiple times by existing attention algorithms such as PagedAttention (Kwon et al., 2023) and FlashAttention (Dao et al., 2022; Dao, 2023), each for an individual request in the batch. This severely slows down LLM inferences, which are known to be memory-bound (Section 3.2). To eliminate such redundant memory access, we propose RelayAttention, an exact algorithm to compute causal attention based on a mathematical reformulation of it. The key idea of RelayAttention is to group the matrix-vector multiplications corresponding to the system prompt into matrix-matrix multiplications, which allow loading the hidden states of the system prompt from DRAM exactly once for all request tokens in a batch (Section 3.3). We provide an in-depth analysis of the theoretic speedup via redundancy reduction with IO-awareness (Section 3.4). Our empirical results further verify the efficiency: integrating RelayAttention into vLLM (Kwon et al., 2023), an already highly optimized production-level LLM serving system, we still observe up to 2.2× sustainable request rate and 2.0× throughput with the Llama2-7B model for a chatbot workload. Similar efficiency improvements are also observed for several other popular LLMs and are consistent on several data center GPUs. The efficiency gains continue growing with longer system prompts.

Our key contributions can be summarized as:

- We have identified a LLM service bottleneck that has not been studied by existing works: there are highly redundant *memory accesses* caused by long system prompts. We anticipate that our analysis will inspire more works on deep architectures with IO-awareness (Dao et al., 2022; Gu and Dao, 2023).
- We propose RelayAttention, a novel approach to compute exact causal attention. It allows accessing cached hidden states of the system prompt exactly once for a batch of request tokens. We conduct an in-depth analysis of the theoretic speedup brought by RelayAttention.
- We empirically verify the end-to-end efficiency improvement by integrating RelayAttention into vLLM, a production level LLM, and observe non-trivial efficiency gains on several popular LLMs with different GPUs.

2 Related Works

Our approach aims to improve the inference efficiency of transformer-based LLMs (Section 2.1). It is based on extending the widely used Key-Value Cache mechanism (Section 2.2). We also briefly review other techniques for accelerating LLM inference, which may complement ours (Section 2.3).

2.1 Inference of Transformer-based LLMs

The inference of these transformer-based LLMs follows the iterative next-token-prediction paradigm. Specifically, the next token is generated in each time step by attending to all precedent tokens. The generated token is then appended to the end of the current sequence. The generation then continues until a stopping criterion (*e.g.*, the new token is `<eos>`, which indicates the end of the sequence) is met. A basic approach to implementing such a generation procedure is to perform full self-attention with a casual mask over the entire up-to-present

sequence at each time step, just as we do while training the model (Radford et al., 2018). This way, a single generation step takes a quadratic complexity w.r.t. the length of the up-to-present sequence. Next, we will look at how this complexity can be reduced to linear using the Key-Value Cache.

2.2 Key-Value Cache

Based on the observation that historical tokens are not affected by the future ones during LLM decoding, Key-Value (KV) Cache avoids repetitive computation of the hidden key-value pairs (KVs) by caching them on the fly and then reusing the cached KVs in every subsequent steps (Yu et al., 2022; Pope et al., 2023). With KV Cache, in each time step, only a *single token* (*i.e.*, the latest generated one) is used as the query, and the next token is produced by attending to the cached KVs. The generation complexity thus reduces from quadratic to linear w.r.t. the up-to-date sequence length.

Some recent research further accelerates LLM inferences by pruning superfluous KV cache data (Zhang et al., 2023) or compressing it (Liu et al., 2023) to reduce key-value pairs to be cached. However, these approaches introduce algebraic discrepancies between model training and inference. Hence, they may hurt the generation quality and/or require extra finetuning efforts. In contrast, our approach maintains generation quality and is plug-and-play, as it is based on a mathematical reformulation of causal attention . The acceleration of our approach comes from reducing redundant *memory access* of the KV cache. Therefore, it is orthogonal and complementary to prefix sharing in PagedAttention (Kwon et al., 2023), which eliminates redundant *memory footprint* of system prompts, and is unlike PromptCache (Gim et al., 2023), which eliminates the redundant *computation* of the reusable prefix KVs and thus only accelerates the prompt phase (Section 3.2).

2.3 Other Optimizations for LLM Inference

Besides the KV Cache, several other techniques optimize LLM inference in a post-training manner. For example, network quantization techniques can also be applied to LLMs as they are architecture-agnostic, even though they may need some adaptations to improve the generation stability and quality (Frantar et al., 2022; Xiao et al., 2023; Lin et al., 2023). FlashAttention (Dao et al., 2022; Dao, 2023) is another technique to optimize LLMs' throughputs on GPUs by avoiding redundant write/read of attention probability matrix into/from DRAM. A production-level LLM serving system may also include continuous batching (Yu et al., 2022), which enables iteration-level scheduling of requests, and speculative sampling (Chen et al., 2023a; Leviathan et al., 2023), which uses a smaller model to generate a draft and then uses the large model to check and correct it. Our approach can work together with these components with no conflicts.

3 Methdology

In this section, we elaborate on the proposed approach. We begin with a brief preliminary of the hardware utilization of operators in Section 3.1, followed by an analysis of the bottleneck in LLM serving in Section 3.2, which shows that the redundant memory access slows down the inference especially when the system prompt is long. We then introduce RelayAttention, a novel algorithm to compute exact causal attention that allows the elimination of the redundancy in Section 3.3. Finally, we analyze the theoretical acceleration of RelayAttention over existing approaches from the perspective of IO-awareness (Section 3.4).

3.1 Preliminary: The Latency of Operators

To increase the utilization of arithmetic units, modern processors use pipelining to allow concurrent memory access and computation. For a perfectly parallelized operator, which maximizes the overlap of data transfer and computation, the runtime latency is determined by the larger one between total memory access time and total computation time. Given a processor that takes t_m for per-byte access, and t_c for a floating operation on average, the ratio r of the total computation time over the total memory access time for an operator is:

$$r = \frac{t_c \times \#\text{floating operations}}{t_m \times \#\text{byte access}} = I \times \frac{t_c}{t_m}, \quad (1)$$

where I is the arithmetic intensity of the operator:

$$I = \frac{\#\text{floating operations}}{\#\text{byte access}}. \quad (2)$$

When $I < \frac{t_m}{t_c}$, r is less than 1, the operator is memory-bound. This means that the bottleneck of the operator is memory access, and we can accelerate it only if we can reduce the memory access time. The speed of modern GPUs far outpaces the bandwidth of its memory (*i.e.*, $\frac{t_c}{t_m} \ll 1$), and thus it typically requires a high arithmetic intensity to

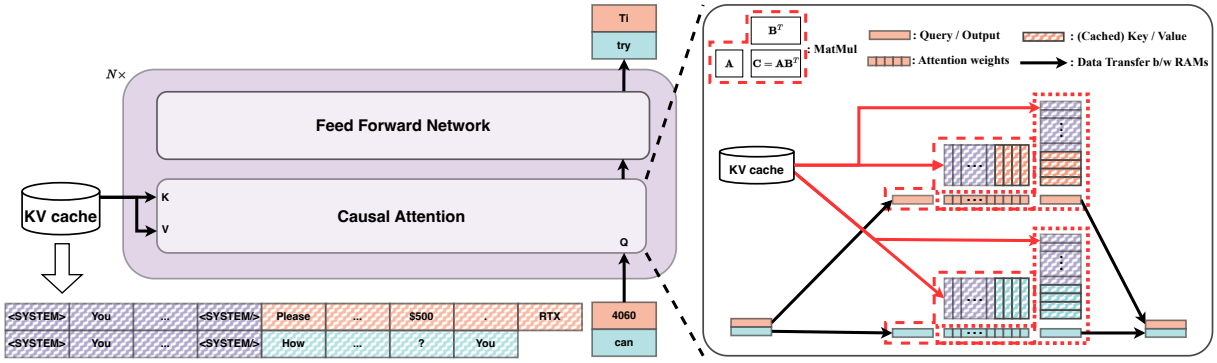


Figure 3: A decoding step during the autoregressive generation phase. On the right side, we provide a closer view of the attention computation with IO-awareness. Note that the floating operations are executed in the fast on-chip SRAM, while the KVs are cached in the slow off-chip DRAM. As highlighted with the dashed boxes and red arrows, (1) the computation mainly involves matrix-vector multiplications; and (2) while being shared by all requests, the system KVs are transferred from DRAM to SRAM multiple times, each for one request.

achieve full utilization of the computing capability (*e.g.*, A100-SXM4 GPU requires at least 38.2).

For a half-precision (2 bytes/element) general matrix multiplication (GEMM) of problem size (m, n, k): $\mathbf{C} = \mathbf{A}\mathbf{B}^T$, where $\mathbf{C} \in \mathbb{R}^{m \times n}$, $\mathbf{A} \in \mathbb{R}^{m \times k}$, $\mathbf{B} \in \mathbb{R}^{n \times k}$, the arithmetic intensity is:

$$I_{gemm} = \frac{2mnk}{2(mk + nk + mn)} < \min\{m, n, k\}. \tag{3}$$

When m, n, k are all large (*e.g.*, > 128), the operation can saturate the utilization of the computing capability due to high arithmetic intensity. This is normally true for linear projection operations in LLM inference, where m is the number of tokens in a batch, k is the input hidden dimension, and n is the output hidden dimension. However, as a special case of GEMMs, the general matrix-vector product (GEMV) operation, in which there is a vector in $\mathbf{A}$ and $\mathbf{B}$, is always memory-bound as $I_{gemv} < 1$. This is the case for casual attention computation with cached KVs, as we will show in Section 3.2.

3.2 Bottleneck of LLM Services

Given a batch of user prompts, the LLM inference is usually divided into two phases: the *prompt phase*, which computes the hidden states of the full prompts (*i.e.*, the concatenation of system prompt and user prompts) and generates the first new tokens; and the *autoregressive generation phase*, which generates *all subsequent tokens* sequentially, one token for each request at a time step. In this work, we focus our investigation on the autoregressive generation phase as it contains the hot spot of response generation[1].

[1]Besides, the prompt phase can effectively saturate GPU utilization as it involves large matrix multiplications.

In Fig. 3, we demonstrate a time step during the autoregressive generation phase, with the batch size assumed to be 2. There are two key observations:

1. **The computation of attention is memory-bound.** This is because the attention computation for a request mainly involves two GEMVs (red dashed boxes in Fig. 3), with an arithmetic intensity lower than 1. It thus requires memory access reduction for acceleration.
2. **There are redundant memory accesses in the typical scenarios where a shared system prompt is prepended to request-specific contexts.** Specifically, the cached key-value pairs of the shared system prompt (system KVs) are read from off-chip DRAM multiple times, each for a request in the batch (red arrows in Fig. 3). Such redundancy becomes a substantial overhead when the system prompt is long.

Section 3.3 proposes the core design of RelayAttention for removing the redundant memory access.

3.3 LLM Serving with RelayAttention

The key idea of RelayAttention is to group multiple matrix-vector multiplications between the batched queries and the cached KVs into single matrix-matrix multiplications, as shown in Fig. 4, allowing system KVs to be read from DRAM exactly once per batch. Algorithm 1 summarizes the algorithm in Pytorch-like pseudo code. It divides the computation of a causal attention layer into three steps: system attention step, context attention step, and relay fusion step. In the system attention and context attention steps, we compute two intermediate attention outputs as if the LLM is prompted by the shared system prompt / request-specific context only. In the relay fusion step, we compute the

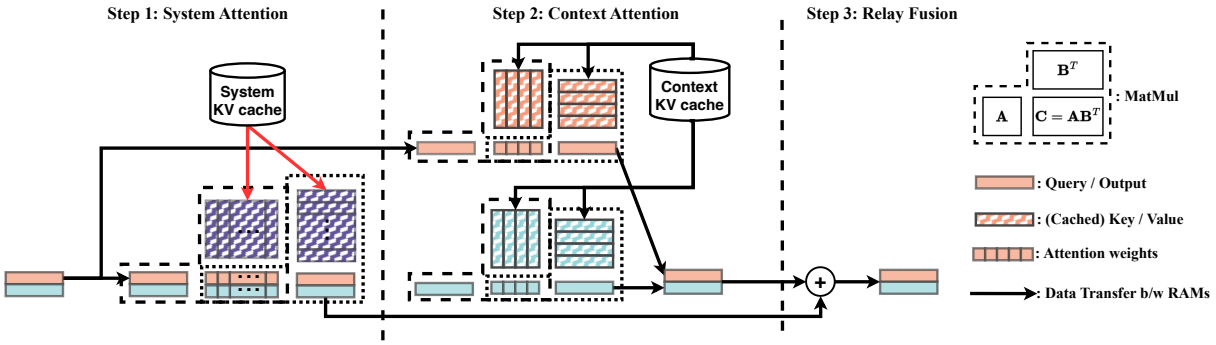


Figure 4: The computation of RelayAttention. It is a mathematical reformulation of casual attention in Fig. 3, but load the System KVs exactly once for a batch of requests (highlighted with red arrows).

Algorithm 1 Pseudocode for RelayAttention.

```
# INPUT:
#   q: query tensor for new inputs, (b, m, h, d)
#   k: key tensor for new inputs, (b, m, h, d)
#   v: value tensor for new inputs, (b, m, h, d)
#   kv_cache: context KVs, (N, 2, b, l-s, h, d)
#   sys_kv_cache: sys. KVs, (N, 2, 1, s, h, d)
#   layer_id: the index of current layer, int
#   l_cache: the length of cached key-value, int
# OUTPUT:
#   o: the output of causal attention

# note: (1) we modified the interface of multi-head
# attention to return the log-sum-exp (lse);
# (2) the order of context attention and system
# attention doesn't matter because of no dependency

# context attention, as if there is no system prompt
# k.size(1) = 1 in autoregressive generation phase
# k.size(1) > 1 in prompt phase
l_new = l_cache + k.size(1)
kv_cache[layer_id, 0, l_cache:l_new, ...] = k
kv_cache[layer_id, 1, l_cache:l_new, ...] = v
o, lse = multihead_attention(
    q, k_cache[layer_id, 0, :l_new, ...],
    v_cache[layer_id, 1, :l_new, ...],
    casual_mask=True)
# system attention
bsz, len, nhead, dim = q.size()
q1 = q.view(1, bsz*len, nhead, dim)
k_sys, v_sys = sys_kv_cache[layer_id].unbind(dim=0)
o_sys, lse_sys = multihead_attention(
    q1, k_sys, v_sys)
o_sys = o_sys.view(bsz, len, nhead, dim)
lse_sys = lse_sys.view(bsz, len, nhead, 1)
# relay fusion
alpha_sys = 1 / (1 + exp(lse - lse_sys))
alpha_usr = 1 - alpha_sys
o = o * alpha_usr + o_sys * alpha_sys
```

final output as a convex combination of the two intermediate outputs . Next, we show that RelayAttention is computing a mathematical reformulation of casual attention.

Without loss of generality, we consider a single sequence in the batch and a single attention head. Formally, given an on-the-fly sequence R at generation step t, we divide it into three segments (in order): (1) the system prompt of length s, (2) the user prompt of length u, and (3) the response generated by the LLM of length $t-1$. Let $\mathbf{k}_i, \mathbf{v}_i \in \mathbb{R}^d$ denote the hidden key, value embedding of the token at position $i \le l = s+u+t$, and $\mathbf{q}_t \in \mathbb{R}^d$ denotes the hidden query embedding in the current step. The casual attention output $\mathbf{o}_t$ is defined as:

$$\begin{aligned}\mathbf{o}_t &= \text{Attention}(\mathbf{q}_t, \{\mathbf{k}_i\}_{i=1}^{l}, \{\mathbf{v}_i\}_{i=1}^{l}) \\ &= \sum_{j=1}^{l} \frac{\exp(\mathbf{q}_t \mathbf{k}_j^T)}{\sigma_t^{1\to l}} \mathbf{v}_j, \end{aligned} \tag{4}$$

where $\sigma_t^{b\to e} = \sum_{j=b}^{e} \exp(\mathbf{q}_t \mathbf{k}_j^T)$ is the sum-exp between the start position b and end position $e > b$, associated with $\mathbf{q}_t$. By splitting the summation in Eq. (4) at position s, which is the end system prompt, we have:

$$\mathbf{o}_t = \sum_{j=1}^{s} \frac{\exp(\mathbf{q}_t \mathbf{k}_j^T)}{\sigma_t^{1\to l}} \mathbf{v}_j + \sum_{j=s+1}^{l} \frac{\exp(\mathbf{q}_t \mathbf{k}_j^T)}{\sigma_t^{1\to l}} \mathbf{v}_j. \tag{5}$$

Consider the first term on the right side of Eq. (5). As it is close to the Attention$(\cdot,\cdot,\cdot)$ operation in Eq. (4), with only a difference in the numerator, it can be rewritten as a rescaled attention:

$$\frac{\sigma_t^{1\to s}}{\sigma_t^{1\to l}} \sum_{j=1}^{s} \frac{\exp(\mathbf{q}_t \mathbf{k}_j^T)}{\sigma_t^{1\to s}} \mathbf{v}_j. \tag{6}$$

This rescaling can also be applied to the second term on the right side of Eq. (5), and thus $\mathbf{o}_t$ is a convex combination of two scaled attention terms:

$$\begin{aligned}\mathbf{o}_t = &\alpha_t^{sys} \text{Attention}(\mathbf{q}_t, \{\mathbf{k}_i\}_{i=1}^{s}, \{\mathbf{v}_i\}_{i=1}^{s}) + \\ &\alpha_t^{ctx} \text{Attention}(\mathbf{q}_t, \{\mathbf{k}_i\}_{i=s+1}^{l}, \{\mathbf{v}_i\}_{i=s+1}^{l}), \end{aligned} \tag{7}$$

where $\alpha_t^{sys} = \frac{\sigma_t^{1\to s}}{\sigma_t^{1\to l}}$ and $\alpha_t^{ctx} = \frac{\sigma_t^{s+1\to l}}{\sigma_t^{1\to l}} = 1 - \alpha_t^{sys}$ are the combination coefficients.

Discussion. Back to the view of a batch, the first term in Eq. (7) for all concurrent requests, namely *system attention*, can be grouped to use large matrix multiplications. This essentially eliminates the

redundant access of system KVs as shown in Fig. 4. In practice, as the sum of exponentials $\sigma_t^{b\to e}$ is not numerically stable to compute directly, we use the log-sum-exp trick to return $\log(\sigma_t^{b\to e})$ in attention computation, and the computation of α_t^{sys} is reformulated accordingly in Algorithm 1. While reformulating the casual attention, we did not assume step $t \neq 1$. This means that RelayAttention is also applicable to the prompt phase, where the input of a request is not a single token generated in the last step but contains multiple tokens from the user prompt, as reflected in Algorithm 1.

Peripheral adaptations. There are two major adaptations needed to make RelayAttention work better within existing inference systems. First, instead of using a single KV cache for both the system prompt and the request-specific context, we use a separate *system KV cache* to store system KVs and fill it offline before model serving. This can be viewed as a combination of prefix sharing in PagedAttention, which eliminates redundant memory footprint of system KVs, and PromptCache (Gim et al., 2023), which eliminates redundant computation in the *prompt phase*. Second, as the system KVs are already computed offline, we add an offset of s (*i.e.*, the length of the system prompt) in the position of those request-specific context tokens to make sure of correct position embedding.

3.4 Theoretical Speedup

In this section, to derive the theoretical speedup of RelayAttention by the memory access reduction, we analyze the memory access during the attention computation of an autoregressive generation step.

Without RelayAttention, given a batch of b request tokens, the number of elements n to transfer between DRAM and SRAM is:

$$n = \underbrace{bd}_{\text{queries}} + \underbrace{b(s+c)d}_{\text{cached KVs}} + \underbrace{bd}_{\text{outputs}}, \tag{8}$$

where d is the embedding dimension, s is the length of the shared system prompt, and c is the length of request-specific context. With RelayAttention enabled, the number of elements to access n' is:

$$n' = \underbrace{(bd + sd + bd)}_{\text{system attetion}} + \underbrace{(bd + bcd + bd)}_{\text{context attention}} + \underbrace{3bd}_{\text{relay fusion}}. \tag{9}$$

Therefore, the speedup p is:

$$p = \frac{n}{n'} = \frac{s + c + 2}{s/b + c + 7}. \tag{10}$$

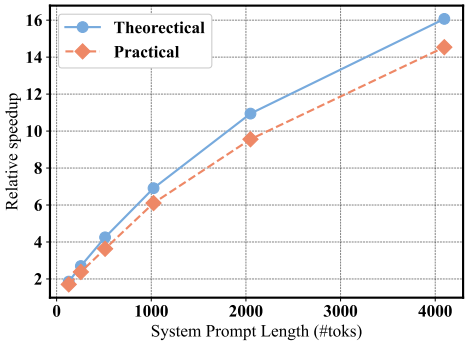


Figure 5: The theoretical and practical speedups for casual attention computation with RelayAttention. We set the batch size to 32 and context length to 128, and plot the speedup w.r.t. the length of the system prompt. A40 GPU is used.

In Fig. 5, we plot the speedup brought by using RelayAttention. The small gap between the practical and theoretical curves verifies our analysis.

Though the speedup of standalone RelayAttention can be analyzed, it is still a question of how an end-to-end LLM serving system can benefit from RelayAttention. In Section 4, we provide an empirical study to answer this question.

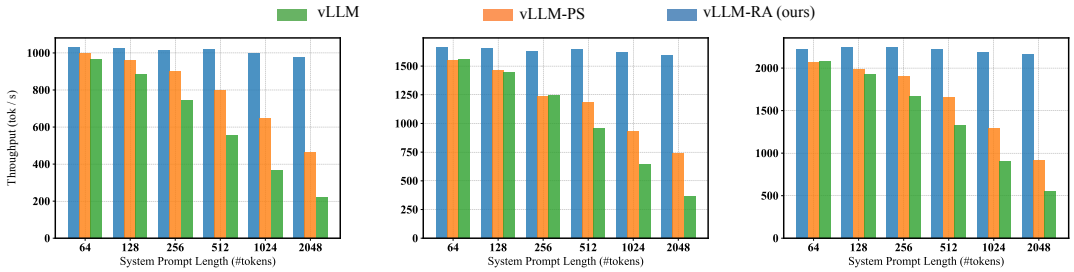
4 Experiments

In this section, we conduct experiments to answer the question of how much our approach can help an end-to-end LLM serving system. We provide the experimental setup in Section 4.1. Our major evaluation is conducted with consideration of two scenarios: noninteractive batch processing (Section 4.2) and interactive service (Section 4.3). We use the Llama2-7B model (Touvron et al., 2023) for evaluation unless stated otherwise. We demonstrate the improvement for more models in Section 4.4.

4.1 Experimental Setup

Data. Two datasets are used in our evaluation: ShareGPTv3 (ShareGPT, 2023) and MMLU (Hendrycks et al., 2021). SharedGPTv3 (ShareGPT, 2023) contains 53k real conversions between users and ChatGPT (OpenAI, 2022). MMLU is a benchmark for measuring massive multitask language understanding in few-shot settings. It consists of 57 tasks covering various subjects and domains, such as mathematics, history, law, and medicine. Each subject/task contains a series of single-choice questions, and 5 extra questions with answers (as few-shot examples). The statistics of the benchmarking data are summarized in Table 1.

Hardware. Our experiments involve three GPUs:



(a) Nvidia A40 throughput (b) A100-PCIE-40GB througput (c) A100-SXM4-80GB throughput

Figure 6: Throughput w.r.t. system prompt length during the noninteractive processing of ShareGPTv3 dataset.

	Sys. prompt len.	User prompt len.	Generation len.
ShareGPTv3	64-2048	4-1024	4-2013
MMLU	379-2895	55-1219	32

Table 1: Data for benchmarking. Lengths are measured in token.

	Memory	Mem. Band.	FP16 Peak F.	Price
A40	48 GB	696 GB/s	37.4 TFLOPs	$0.40/hr
A100-PCIE-40GB	40 GB	1,555 GB/s	77.9 TFLOPs	$0.90/hr
A100-SXM4-80GB	80 GB	2,039 GB/s	77.9 TFLOPs	$1.84/hr

Table 2: The specifications of the GPUs used in our experiments. Prices are from vast.ai.

A40, A100-PCIE-40GB, and A100-SXM4-80GB. However, A40 is used unless stated otherwise. The hardware specifications are listed in Table 2.

Three Approaches used for comparison:

- **vLLM** [2]: a state-of-the-art open-source LLM inference system designed for high throughput LLM serving. Note that the core component of vLLM, PagedAttention (Kwon et al., 2023), allows *storing* the shared system KVs exactly once by the prefix sharing technique mentioned in their paper, but this technique is not included in the public code release. Considering the importance to save memory for a higher concurrency, we implement a stronger baseline, vLLM-PS as specified below.
- **vLLM-PS**: the augmented version of vLLM implemented by us. It integrates not only prefix sharing but also PromptCache (Gim et al., 2023), which precomputes the system KVs and reuses them to mitigate the burden of the *prompt phase*. Therefore, vLLM-PS eliminates both redundant *memory footprint* and unnecessary *computations* of system KVs.
- **vLLM-RA (ours)**: the modfied vLLM with our RelayAttention integrated. Compared with vLLM-PS, this version further eliminates the redundant *memory accesses* of system KVs, as discussed in Section 3.3.

[2] https://github.com/vllm-project/vllm

	Accuracy	GPU	vLLM	vLLM-PS	vLLM-RA
1-shot	37.6%	A100-80G	502	336(↓33%)	306(↓39%)
		A40-48G	1012	675(↓33%)	621(↓39%)
3-shot	41.3%	A100-80G	851	378(↓55%)	311(↓63%)
		A40-48G	1751	752(↓57%)	629(↓64%)
5-shot	43.2%	A100-80G	1308	432(↓67%)	316(↓76%)
		A40-48G	2660	850(↓68%)	641(↓76%)

Table 3: MMLU few-shot acc. and processing time (s).

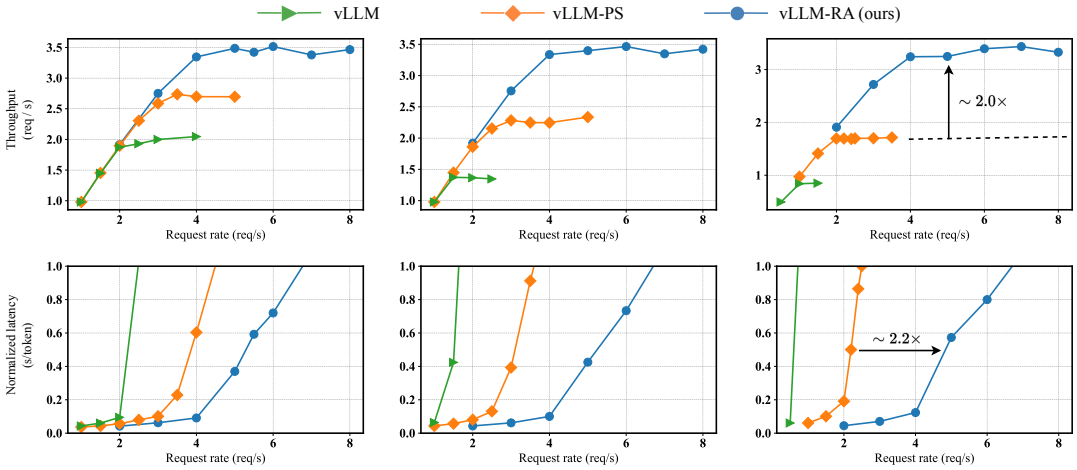
4.2 Noninteractive Batch Processing

For the non-interactive batch processing scenarios where users just submit their jobs to the LLM services and harvest the processing results later, we consider the throughput (number of tokens per second) and processing time as the key metrics.

We plot the throughputs w.r.t. the length of system prompt for processing ShareGPTv3 on the three GPUs in Fig. 6. For vLLM, the throughputs degrade heavily as the system prompt becomes long for two reasons: (1) the system prompt occupies too much memory, and thus heavily limits the batch size/concurrency of decoding; (2) it takes too much time to handle long system prompts during causal attention computation. With prefixing sharing, vLLM-PS solves the first problem and achieves up to 108% improvement on the throughput. Our vLLM-RL further solves the second problem and increases the throughput to $1.06\times$ to $4.36\times$ of vLLM. Table 3 shows results of the few-shot test on MMLU. We can see that using a long system prompt to include more examples is crucial for improving accuracy. In the case of the 5-shot test, our vLLM-RA provides a 76% reduction of processing time on both A40 and A100-SXM4-80GB GPUs.

4.3 Interactive Serving

An important LLM application is chatbots (OpenAI, 2022; Google, 2023a), in which interactive LLM services are typically provided. Unlike the noninteractive scenario, though we still expect a high throughput for good hardware utilization, we also care about the normalized latency (*i.e.*, average



(a) System prompt length = 512 (b) System prompt length = 1024 (c) System prompt length = 2048

Figure 7: Benchmark interactive serving with requests sampled from the ShareGPTv3 dataset.

per-token latency), which is crucial for user experience. Following PagedAttention (Kwon et al., 2023), we sample 1000 requests from the ShareGPTv3 dataset to benchmark the efficiency. The request arrival times are generated using Poisson distribution with different request rates.

As shown in Fig. 7, as the request rate increases, the throughput grows gradually until reaching a maximum. In contrast, the latency remains low at the beginning and then goes up steeply when the highest throughput is achieved. Around the latency of 0.5s/token, where the user experience and hardware utilization is balanced, vLLM-RA sustains higher request rates than both vLLM and vLLM-PS with clear margins (up to $\sim 2.2\times$ when the system prompt length is 2048).

4.4 The Improvement for More Models

To verify the efficiency improvement for more models, we choose several other popular LLMs such as Llama2-13B, Llama-30B, Phi-2 (Microsoft, 2023b), and Mistral-7B (Jiang et al., 2023) to run the noninteractive batch processing of ShareGPTv3. As shown in Table 4, vLLM-RA also provides consistent improvements for these LLMs.

	vLLM	vLLM-PS	vLLM-RA
system prompt length = 512			
Llama2-13B	0.99	1.44 (↑45%)	1.71 (↑73%)
Llama-30B†	2.15	3.01(↑40%)	3.65(↑70%)
Phi-2 (2.7B)	5.03	6.29 (↑25%)	8.85(↑76%)
Mistral-7B	3.68	5.40 (↑47%)	5.90(↑60%)
system prompt length = 1024			
Llama2-13B	0.66	1.23(↑86%)	1.69(↑156%)
Llama-30B†	1.52	2.55(↑68%)	3.64(↑139%)
Phi-2 (2.7B)	3.54	4.82(↑36%)	8.76(↑147%)
Mistral-7B	2.60	4.92(↑89%)	5.85(↑125%)

Table 4: Throughput (req/s) of different models during the batch processing of the ShareGPTv3 dataset. †: the 30B model is hosted on two A100-SXM4-80GB GPUs.

5 Limitations and Future Work

The limitations of RelayAttention can be reflected by the theoretical speedup (Eq. (10)). First, it helps *batched* inference ($b > 1$). The larger the batch size, the more efficient RelayAttention is. When there is only one request, which is the typical case on device-side applications, RelayAttention does not help. Therefore, RelayAttention is suitable for cloud-serving scenarios. Second, when the request-specific contexts (including user prompts and responses) are long (*e.g.*, $2\times$ longer than the shared system prompt), the computation time is dominated by the processing of them; thus the efficiency gain will diminish. However, as the context length has a long-tailed distribution in many applications (*e.g.*, chatbots), where the majority of user prompts and responses are short, the efficiency gain brought by RelayAttention is still considerable. In future work, we will explore more applications by customizing a LLM with long system prompts.

6 Conclusion

In this paper, we have identified a bottleneck of using long system prompts in LLM services: there are highly redundant memory accesses corresponding to those system KVs. We have proposed RelayAttention to compute exact causal attention while removing the redundant memory accesses. An analysis of the theoretical speedup of RelayAttention is provided. Extensive experiments over different GPUs, models, and datasets empirically verify the efficiency gains brought by RelayAttention.

References

Anthropic. 2023. https://www.anthropic.com/index/100k-context-windows.

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. 2023a. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Yukang Chen, Shengju Qian, Haotian Tang, Xin Lai, Zhijian Liu, Song Han, and Jiaya Jia. 2023b. Longlora: Efficient fine-tuning of long-context large language models. *arXiv preprint arXiv:2309.12307*.

Jiaxi Cui, Zongjian Li, Yang Yan, Bohua Chen, and Li Yuan. 2023. Chatlaw: Open-source legal large language model with integrated external knowledge bases. *arXiv preprint arXiv:2306.16092*.

Tri Dao. 2023. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*.

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

DeepSeek-AI, :, Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, Huazuo Gao, Kaige Gao, Wenjun Gao, Ruiqi Ge, Kang Guan, Daya Guo, Jianzhong Guo, Guangbo Hao, Zhewen Hao, Ying He, Wenjie Hu, Panpan Huang, Erhang Li, Guowei Li, Jiashi Li, Yao Li, Y. K. Li, Wenfeng Liang, Fangyun Lin, A. X. Liu, Bo Liu, Wen Liu, Xiaodong Liu, Xin Liu, Yiyuan Liu, Haoyu Lu, Shanghao Lu, Fuli Luo, Shirong Ma, Xiaotao Nie, Tian Pei, Yishi Piao, Junjie Qiu, Hui Qu, Tongzheng Ren, Zehui Ren, Chong Ruan, Zhangli Sha, Zhihong Shao, Junxiao Song, Xuecheng Su, Jingxiang Sun, Yaofeng Sun, Minghui Tang, Bingxuan Wang, Peiyi Wang, Shiyu Wang, Yaohui Wang, Yongji Wang, Tong Wu, Y. Wu, Xin Xie, Zhenda Xie, Ziwei Xie, Yiliang Xiong, Hanwei Xu, R. X. Xu, Yanhong Xu, Dejian Yang, Yuxiang You, Shuiping Yu, Xingkai Yu, B. Zhang, Haowei Zhang, Lecong Zhang, Liyue Zhang, Mingchuan Zhang, Minghua Zhang, Wentao Zhang, Yichao Zhang, Chenggang Zhao, Yao Zhao, Shangyan Zhou, Shunfeng Zhou, Qihao Zhu, and Yuheng Zou. 2024. Deepseek llm: Scaling open-source language models with longtermism.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2022. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*.

In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2023. Prompt cache: Modular attention reuse for low-latency inference. *arXiv preprint arXiv:2311.04934*.

GitHub. 2022. Github copilot. https://github.com/features/copilot.

Google. 2023a. https://bard.google.com.

Google. 2023b. Gemini - google deepmind. https://deepmind.google/technologies/gemini.

Albert Gu and Tri Dao. 2023. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021. Measuring massive multitask language understanding. *Proceedings of the International Conference on Learning Representations (ICLR)*.

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7b. *arXiv preprint arXiv:2310.06825*.

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

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

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, and Song Han. 2023. Awq: Activation-aware weight quantization for llm compression and acceleration. *arXiv preprint arXiv:2306.00978*.

Yuhan Liu, Hanchen Li, Kuntai Du, Jiayi Yao, Yihua Cheng, Yuyang Huang, Shan Lu, Michael Maire, Henry Hoffmann, Ari Holtzman, et al. 2023. Cachegen: Fast context loading for language model applications. *arXiv preprint arXiv:2310.07240*.

Microsoft. 2023a. https://www.microsoft.com/en-us/windows/copilot-ai-features.

Microsoft. 2023b. https://www.microsoft.com/en-us/research/blog/phi-2-the-surprising-power-of-small-language-models/.

John J Nay, David Karamardian, Sarah B Lawsky, Wenting Tao, Meghana Bhat, Raghav Jain, Aaron Travis Lee, Jonathan H Choi, and Jungo Kasai. 2023. Large language models as tax attorneys: A case study

in legal capabilities emergence. *arXiv preprint arXiv:2306.07075*.

OpenAI. 2021. https://openai.com/research/triton.

OpenAI. 2022. https://openai.com/blog/chatgpt.

OpenAI. 2023a. https://openai.com/blog/custom-instructions-for-chatgpt.

OpenAI. 2023b. GPT-4 technical report. *CoRR*, abs/2303.08774.

Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2023. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*, 5.

Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training.

Laila Rasmy, Yang Xiang, Ziqian Xie, Cui Tao, and Degui Zhi. 2021. Med-bert: pretrained contextualized embeddings on large-scale structured electronic health records for disease prediction. *NPJ digital medicine*, 4(1):86.

ShareGPT. 2023. https://sharegpt.com/.

Ethan Steinberg, Ken Jung, Jason A Fries, Conor K Corbin, Stephen R Pfohl, and Nigam H Shah. 2021. Language models are an effective representation learning technique for electronic health record data. *Journal of biomedical informatics*, 113:103637.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2023. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*, pages 38087–38099. PMLR.

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. 2023. H _2 o: Heavy-hitter oracle for efficient generative inference of large language models. *arXiv preprint arXiv:2306.14048*.

A Implementation of RelayAttention

Reformulation of relay fusion. As mentioned in Section 3.3, we use the log-sum-exp trick to handle the numerical instability of the denominator in Softmax operation. The combination coefficient for the system attention term in Eq. (7), α_t^{sys}, is reformulated accordingly as:

$$\begin{aligned}\alpha_t^{sys} =& \frac{\sigma_t^{1\to s}}{\sigma_t^{1\to l}} = \frac{\sigma_t^{1\to s}}{\sigma_t^{1\to s} + \sigma_t^{s+1\to l}} \\ =& \frac{\exp(\beta_t^{1\to s})}{\exp(\beta_t^{1\to s}) + \exp(\beta_t^{s+1\to l})} \\ =& \frac{1}{1 + \exp(\beta_t^{s+1\to l} - \beta_t^{1\to s})},\end{aligned} \tag{11}$$

where

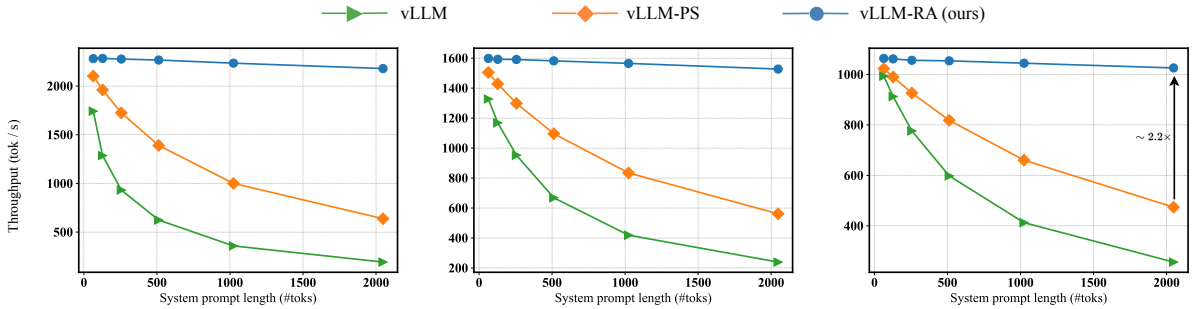
$$\beta_t^{b\to e} = \log(\sigma_t^{b\to e}) = \log(\sum_{j=b}^{e} \exp(\mathbf{q}_t \mathbf{k}_j^T)) \tag{12}$$

is the log-sum-exp.

Implementation details. RelayAttention can be built up on existing efficient attention kernels with minimal adaptations. For the system attention involving the system prompt of non-growing static length, we use off-the-shelf FlashAttention kernels (Dao, 2023), which natively return the log-sum-exp required for computation of combination coefficients in Eq. (7). For the context attention that needs to handle the growing request-specific contexts, we use PagedAttention (Kwon et al., 2023) kernels for efficient memory management and modify these kernels to return log-sum-exp. We implement a single fused kernel with OpenAI Triton (OpenAI, 2021) for the relay fusion step, which involves multiple element-wise operations.

B System Level Design of vLLM-RA

As mentioned in Section 3.3, it is easy to integrate RelayAttention into existing inference system with the replacement of attention computation function and several peripheral adaptations. In Fig. 9, we summarize the system level design of vLLM-RA in a comparison with vLLM.



(a) user prompt len. 64, generation len. 128 (b) user prompt len. 128, generation len. 256 (c) user prompt len. 256, generation len. 512

Figure 8: Throughput w.r.t. system prompt length with synthetic workloads.

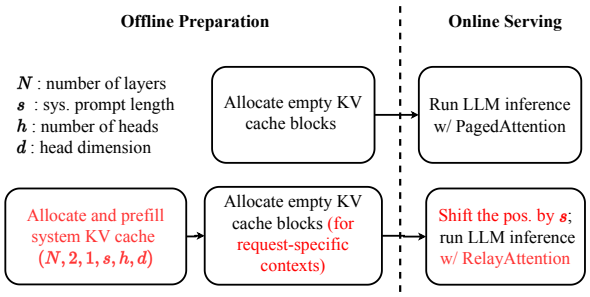


Figure 9: Comparison of the system level design of vLLM (top) and vLLM-RA (bottom). The modifications of vLLM-RA are highlighted in red.

C More Information of The Datasets

The ShareGPTv3 dataset contains both user prompts and LLM responses. The distributions of the length are plotted on the top of Fig. 10. We use synthesized system prompts during benchmarking with this dataset.

For the MMLU dataset, we use the provided few-shot examples as system prompts and the questions as user prompts. The generation length is set to 32 and we extract the answer in A, B, C, D as the first capital letter in the responses. The length distributions of system prompts and user prompts are shown in Fig. 10 bottom.

D Benchmark with Synthetic Workloads

In the section, we benchmark the efficiency with synthetic workloads, where the user prompt length and the generation length are both fixed for all requests. Though this is far from real-world scenarios, it is useful to test the limit of an LLM serving system because such perfectly length-aligned requests eliminate the burden of scheduling. We adopt three combinations of user prompt length and generation length, (64, 128), (128, 256), and (256, 512) for benchmarking, and plot the trend of throughput w.r.t. the system prompt lenth in

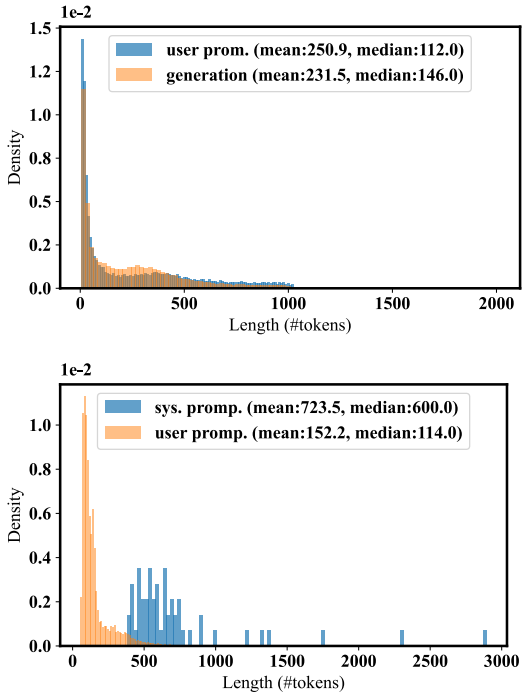


Figure 10: Distribution of the two datasets: ShareGPTv3 (top) and MMLU (bottom).

Fig. 8. Notably, in the most challenging case where the request-specific contexts have a length of $256 + 512 = 768$, RelayAttention still provides an up to $2.2\times$ speedup when the system prompt length is 2048.