Stateful Large Language Model Serving with Pensieve

Lingfan Yu
New York University
lingfan.yu@nyu.edu

Jinkun Lin
New York University
jinkun.lin@nyu.edu

Jinyang Li
New York University
jinyang@cs.nyu.edu

Abstract

Large Language Models (LLMs) are wildly popular today and it is important to serve them efficiently. Existing LLM serving systems are stateless across requests. Consequently, when LLMs are used in the common setting of multi-turn conversations, a growing log of the conversation history must be processed alongside any request by the serving system at each turn, resulting in repeated processing.

In this paper, we design *Pensieve*, a system optimized for multi-turn conversation LLM serving. *Pensieve* maintains the conversation state across requests by caching previously processed history to avoid duplicate processing. *Pensieve*'s multi-tier caching strategy can utilize both GPU and CPU memory to efficiently store and retrieve cached data. *Pensieve* also generalizes the recent PagedAttention kernel to support attention between multiple input tokens with a GPU cache spread over non-contiguous memory. Our evaluation shows that *Pensieve* can achieve 1.14-3.0× the throughput of vLLM and TensorRT-LLM and significantly reduce latency.

CCS Concepts: • **Software and its engineering** → **Virtual memory**; • **Computer systems organization** → *Data flow architectures.*

Keywords: LLM Serving, Multi-turn Conversations, Cache

1 Introduction

The world has recently witnessed the fast expansion of Large Language Models (LLMs). The most popular use of LLM is for chatbots, with applications like ChatGPT demonstrating an astounding capability of following instructions and interacting with humans as a virtual assistant. Other LLM-backed applications include writing code, responding to emails, doing literature reviews, etc. As LLM continues its explosive growth, it is imperative to develop fast and efficient LLM serving systems.

An LLM is an autoregressive DNN model based on the Transformer [48] architecture. The model iteratively predicts the next output token based on the current context which includes the sequence of input prompt tokens followed by output tokens generated in the previous iterations. LLMs

require very expensive computation due to two factors. One, LLMs have huge parameter sizes (10s or 100s of billions) and have a trend of growing even larger. Two, LLMs need to support a large context size (2K to 32K tokens) to be useful. There has been much related work to improve the performance of LLM inference/serving from various angles, including better batching [52], operation fusion [10], better GPU memory utilization [27], faster output generation [6, 28], low rank adaptation [21] and quantization [12] (see §7). In contrast to these works, in this paper, we take a step back and examine inefficiencies that arise in the setting of a specific but very popular LLM use case today, aka as a multi-turn conversational chatbot.

In the conversational setup, the user and the chatbot are engaged in a dialogue that may last many rounds. In order for the chatbot not to "lose memory" of what has been said so far when responding, the cumulative history of the dialogue must be part of the context for LLM's autoregressive generation. As existing LLM serving systems are stateless across requests, one must prepend a growing log of conversation history alongside each new request as the input prompt to be processed from scratch. This causes much duplicate processing for multi-turn conversations.

How to avoid duplicate processing of the chat history? To do so, the serving system can save any previously processed context data in the form of token embeddings. When new requests from the same conversation arrive, the saved context data can be re-used and subsequently augmented. This can be done with best effort. Essentially, the serving system is allowed to keep some cached state containing previously processed context across requests. Doing so enables the serving system to exploit the opportunity that when users are actively chatting with an AI chatbot, follow-up requests usually arrive within a reasonably short time period to leverage the cached state.

Caching state across requests is straightforward at the high level, but several challenges remain to make it really work. First, where to save the data? Keeping it in the GPU is the fastest, but is very constrained by the relatively small GPU memory size. Putting it on disk would incur much longer load latency, hurting the user experience. A two-tier caching solution spanning both GPU and CPU memory is promising, but care must be taken to cope with each tier's capacity limit and to swap in/out saved context data from/to the GPU without damaging performance. Second, how to reuse saved context data efficiently when processing a new request? Furthermore, some parts of the saved context can

---

be dropped due to the cache limit. How to handle partially saved context by recomputing what has been dropped?

In this paper, we design a stateful LLM serving system, called Pensieve, to address the aforementioned challenges. Pensieve saves a conversation's processed context in a two-tier GPU-CPU cache. It evicts cached data to the next tier (or discards it), preferring conversations that have been inactive for longer and/or those that are cheaper to recompute. The eviction is done at the granularity of a chunk of tokens instead of the whole conversation. Therefore, a conversation's saved context might span both tiers of the cache and may be partially dropped. Pensieve uses ahead-of-time swapping and pipelined transfer to overlap computation with the data movement between cache tiers. Dropped contexts are handled via recomputation. Evicting and restoring cause a conversation's cached context to occupy non-contiguous GPU memory. We develop a new GPU kernel to compute attention [48] between multiple input tokens and cached context residing in non-contiguous memory, which is lacking in existing LLM serving systems. Our kernel is a generalized version of the PagedAttention kernel in vLLM [27].

We have built Pensieve and compared its performance against vLLM [27] and TensorRT-LLM [36], two state-of-the-art serving systems that do not cache state across requests. Experiments show that Pensieve achieves serving throughput that is 1.14-1.70× of vLLM and TensorRT-LLM for small single-GPU models (OPT-13B, Llama 2-13B) and 1.64-3.0× for large four-GPU models (OPT-66B, Llama 2-70B). At moderate serving load, Pensieve also significantly reduces latency.

In summary, this paper makes the following contributions:

- We identify a major inefficiency of existing LLM serving systems when used for multi-turn conversations: a conversation's history context is recomputed with each successive new request in the same conversation.
- We develop Pensieve, a stateful LLM serving system that saves the conversation context in a multi-tier GPU-CPU cache and reuses it across requests to minimize redundant computation. Our system design can efficiently move data between cache tiers and handle partially dropped context via recomputation.
- We build a new attention GPU kernel to compute attention between a new request's multiple input tokens and the saved context scattered in non-contiguous GPU memory. Existing kernels either require contiguous GPU context cache or are restricted to a single input token.
- We evaluate Pensieve using real-world conversation datasets to demonstrate its effectiveness compared to the state-of-the-art stateless serving system.

2 Background

This section provides a brief background on LLMs and how existing systems serve them.

2.1 LLM and the Attention Mechanism

Popular large language models, e.g. GPT-3 [5], OPT [55], Llama [46, 47], are all based on the Transformer architecture [48]. A model consists of many transformer layers, each of which is composed of an attention module, seen as the dashed box Figure 1, and a 2-layer feed-forward network. The model takes as input a sequence of token IDs representing the natural language sentence and feeds them through an embedding layer to obtain a continuous representation (aka embedding) for each token before feeding them through transformer layers. For simplicity, we refer to token embeddings as "tokens", and refer to token IDs as "raw tokens". LLM is autoregressive in the sense that it iteratively predicts the next output token based on the current context which includes the input prompt tokens followed by output tokens generated in the previous iterations.

The success of the Transformer originates from the capability of its attention module. For each layer, the attention module first performs QKV projection, aka linear transformations on its input token embedding (X), to produce three new embeddings Query (Q), Key (K), and Value (V):

$$Q = XW_{query}$$
$$K = XW_{key} \qquad (1)$$
$$V = XW_{value}$$

where W_{query}, W_{key} and W_{value} are trainable weights. We refer to the K and V embeddings as KV-tokens. The module then computes all-to-all attention scores using the dot product between each pair of tokens' query and key:

$$A = \frac{QK^T}{scale} \qquad (2)$$

where scale is a normalization factor. This attention score is then normalized using softmax and used for weighted aggregation of the token embeddings in V:

$$O = softmax(A)V \qquad (3)$$

The above equations show the process of so-called single-head attention. In practice, multi-head attention is used, where Q, K, V produced by Equation (1) are divided into groups, called attention heads. Each attention head independently performs attention as Equation (2) and (3).

2.2 How LLM is Served

The prefill vs. generation phase. To perform inference using an LLM, one needs to keep a KV cache in GPU to avoid recomputation during the autoregressive output generation. Figure 1 shows the typical LLM inference process adopted by systems like FasterTransformer [35], ORCA [52], vLLM [27]. It is divided into two phases: 1) In the prefill phase, all input prompt tokens are processed together to generate K and V (aka KV-tokens) for each layer, and the KV cache is initialized with the resulting KV-tokens. The embedding of the last token from the last layer is used to generate the first output
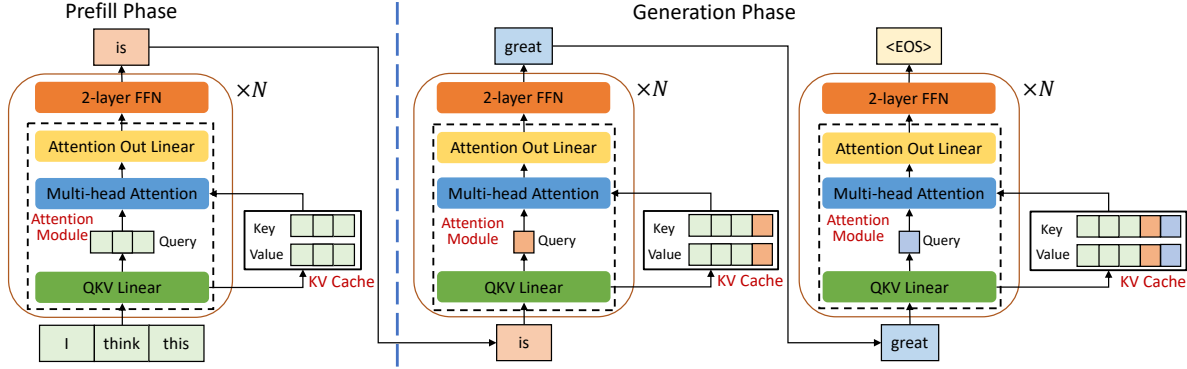
Figure 1. How inference is done for Transformer-based LLMs

token; 2) The generation (also referred to as decoding) phase works iteratively over many steps. In each step, the token generated by the last generation step is processed as a single new input token. Each layer computes the Q,K,V embedding vector for the new token, updates the KV cache, and performs attention using the new token's Q embedding with all KV-tokens in the KV cache.

Iteration-level batching. For LLMs that have variable-sized input and output, the granularity of batching has a huge impact on system throughput and serving latency. If scheduling is performed at the request granularity, executing a batch of requests with different input prompt lengths requires padding tensors to the maximum length and waiting for the request with the longest output to finish. Iteration-level batching strategy, originally proposed by BatchMaker [17] for non-transformer-based sequence-to-sequence models, performs batching at token granularity. ORCA [52] extends this approach to support the LLM workload: whenever a request finishes an iterative generation step, the scheduler checks whether it has reached the end of a sequence and can leave the batch, making room for a request to start its generation phase immediately.

Memory management. For each request, the model performs iterative generation until either the special end-of-sentence token (EOS) is emitted or the preconfigured maximum decoding length is reached. Systems like FasterTransformer [35] and ORCA [52] reserve slots in KV cache for each request based on the maximum decoding size. A more recent system, vLLM [27], can dynamically grow the allocated cache slots for each request and allow these slots to reside in non-contiguous GPU memory. vLLM develops PagedAttention GPU kernel to handle the generation phase with non-contiguous KV cache. Existing serving systems are stateless across requests. In other words, they de-allocate all the cache slots used by a request as soon as it finishes.
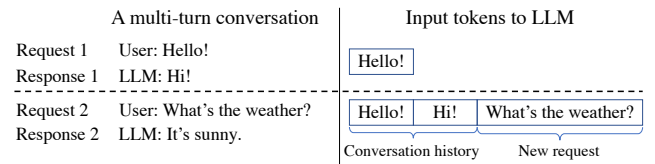

Figure 2. Existing serving systems process a cumulative history repeatedly with each request in a multi-turn conversation.

3 Motivation and Challenges

3.1 Motivation

Existing techniques for serving LLMs mostly focus on improving the inference time or memory efficiency of a single request. We take a step back and examine inefficiencies that arise in a very popular LLM use case today, aka as a multi-turn conversational chatbot.

In a multi-turn conversation, the user engages in multiple rounds of conversations with the chatbot so the underlying LLM needs to be aware of the conversation history to generate an appropriate response. This is done by prepending the cumulative conversation history as raw text to each new request, due to the stateless nature of existing serving systems, as shown in Figure 2. As the interaction between the user and chatbot continues, the conversation history grows, making the cost of the prefill phase overshadow that of the iterative generation phase. Unfortunately, much of the history processing is redundant.

Figure 3 demonstrates the heavy cost of the prompt initiation phase under an artificial workload where each request has 200 new prompt tokens and has varying conversation history sizes. As shown in the figure, the cost of recomputing the conversation history (solid blue line) causes the cost of the prefill phase to soon outgrow the generation phase.

The goal of this project is to minimize redundant computation of the conversation history. This can be done by caching any previously processed embeddings at the serving system and re-using them across requests from the same
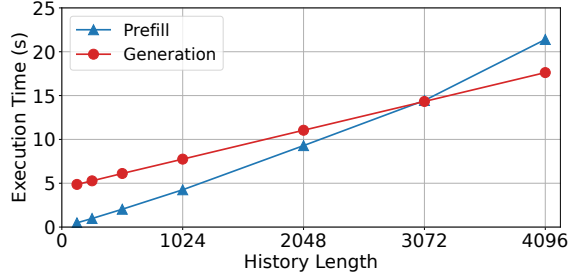
Figure 3. Execution time for a batch of 32 requests performing prompt (32 tokens) prefill and generations for 200 steps.

Figure 4. Execution time of attention operation for a chunk of 32 tokens with different context sizes. Results are normalized by the execution time of non-attention operations in a transformer layer.

Figure 5. Layout of a typical request's KV-token context. The shaded areas, which occur at both ends of the context, mark those tokens that must be processed by the prefill phase.

conversation. More concretely, one can save the KV-tokens in the KV cache belonging to a previous request and only process the new user prompts of the next follow-up request while re-using history embeddings saved in the KV cache.

3.2 Challenges

Limited GPU memory for caching. LLM has very large model parameters, resulting in large KV-tokens. For example, a 13 billion parameter GPT-3 model has 40 layers and a hidden size of 5120. Assuming the use of 16-bit half-precision numbers, storing each KV-token takes 2 * 40 (layer) * 5120 (units/layer) * 2 (bytes/unit) = 0.78MB memory space. Given the limited GPU memory capacity, depending on the history lengths, only a few dozen or hundreds of conversation histories can be kept in the GPU. Therefore, we must extend our cache space to use the more abundant CPU memory.

Token-level cache management and recovery. When using a multi-tier GPU-CPU cache, the serving system needs to swap cached history from GPU to CPU and vice versa. Swapping at the coarse granularity of an entire conversation history is sub-optimal; not only does it utilize the cache space inefficiently but also it incurs large swapping latency. Thus, we decide to swap at the granularity of individual tokens. Specifically, in order to make room for the processing of new requests, the serving system chooses certain cached KV-tokens to swap from GPU to CPU. Later, it also needs to restore the swapped out KV-tokens from CPU to GPU.

When the system is under CPU memory pressure, some cached KV-tokens need to be dropped and re-computed later when needed. We note that, although each token occupies the same amount of memory space, the recomputation cost of each token is different due to the nature of causal attention computation. Specifically, tokens appearing later in the context sequence require more computation than earlier ones because tokens "attend to" all preceding but not succeeding tokens. Figure 4 shows the execution time of the attention operator for 32 tokens with a prompt context of varying sizes; the execution time shown has been normalized against the rest of inference time (aka the sum of execution times
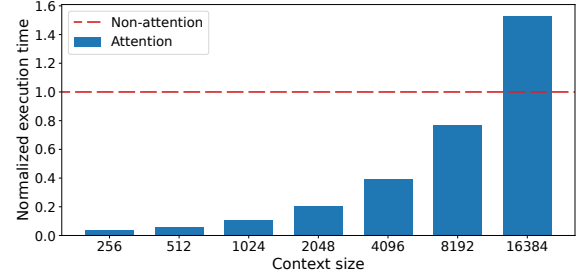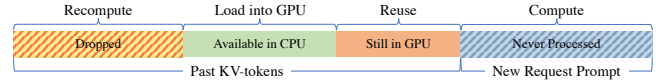
for all other non-attention operators). As can be seen in Figure 4, the cost of attention grows linearly with context size. Thus, when deciding which tokens to discard to reclaim CPU memory, it is more preferable to drop the leading tokens of a conversation history.

Dropping KV-tokens from the leading end of a conversation brings additional complexities. Figure 5 illustrates the layout of a typical request from a continuing conversation in its prefill phase. The request's context can be viewed as composed of four segments: 1) the first and earliest segment corresponds to tokens that have been dropped from the CPU cache and must be recomputed. 2) the second segment corresponds to tokens that reside in the CPU cache and will be fetched into the GPU. 3) the third segment contains tokens residing in the GPU cache. 4) the fourth and latest segment contains the raw tokens corresponding to this request's new prompt. As we can see, both the first and fourth segment requires computation. However, such separation of computation at both ends of the context breaks the assumption of all existing attention kernels that the input tokens belong in a consecutive context region in the prefill phase.

Handling non-contiguous KV cache. Existing systems [27, 52] batch requests separately for the prefill and generation phase so that they can use existing high-performance attention kernels [10, 39] for the prefill phase. Unfortunately, we cannot simply adopt such a design in our setting. This is because existing attention kernels for prefill assume a KV cache with contiguous memory. However, in order to support the swapping of KV-tokens between GPU and CPU, it is more efficient to allow KV-tokens to reside in non-contiguous GPU memory regions. Although vLLM [27] has developed the
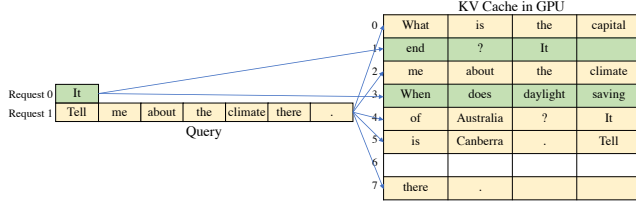
Figure 6. Multi-token attention with non-contiguous KV cache. The batch contains two requests, whose tokens (Query representations) are shown on the left. The right side shows the requests' context (KV-tokens) which reside in non-contiguous GPU memory (chunk 3, 1 for request 0 and chunk 0, 4, 5, 2, 7 for request 1).

PagedAttention kernel to handle non-contiguous KV cache, it is designed to be solely used in the generation phase, because it limits each request in the batch to have exactly one input token. As each request has more than one token in the prefill phase, one cannot simply use PagedAttention for prefill. A naive hack is to process the new prompt one token at a time, in the same manner as iterative generation, so that the PagedAttention kernel can be applied. But this method gives up the parallelization opportunity brought by the extra query token dimension in the prefill phase. Thus, to achieve efficient GPU computation, we must address the challenge of supporting non-contiguous KV cache during prefill. Doing so also brings extra benefit: as both the prefill phase and generation phase can compute using the same non-contiguous KV cache, we can handle requests in different phases together in the same batch. Figure 6 illustrates the desired Multi-token attention kernel for computing attention for a batch of two requests, one in its generation phase, the other in its prefill phase, over non-contiguous KV cache.

4 System Design

At a high level, we aim to save a conversation's KV-tokens across multiple turns in a multi-tier GPU-CPU cache. To realize the potential performance benefits, we need to make cache swapping and dropped token recomputation efficient, by developing techniques to address the challenges in §3.

4.1 System Overview

Figure 7 shows the system architecture. Pensieve consists of a single scheduler and multiple workers, each of which manages one GPU. The scheduler has two jobs: 1) it is responsible for batching requests for execution, and 2) it ensures that requests in the batch have sufficient GPU memory for execution. For 1), the scheduler performs fine-grained iteration-level batching [17, 27, 52] so that a new request can join the batch with existing requests while the latter is in the process of performing autoregressive generation. For 2), the scheduler manages the allocation of slots in the KV cache and determines when to swap between the GPU and CPU KV cache. Each worker also has two jobs: 1) it invokes
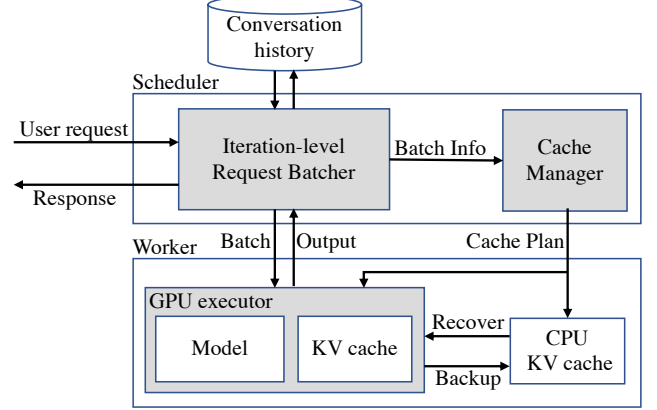

Figure 7. System architecture of Pensieve

GPU kernels to process a batch of requests. 2) it performs the actual data movements between the GPU and CPU KV cache based on the batch's cache plan as determined by the scheduler.

4.2 A unified batch scheduler

Pensieve performs fine-grained iteration-level batching [17, 27, 52]. However, our batching strategy differs from existing LLM serving systems. While ORCA only batches non-attention operations and handles attention operations individually for each request [52], Pensieve and vLLM [27] batch requests for both non-attention and attention operations. However, unlike vLLM which only forms a batch among requests in the same phase and thus processes each of the two types of batches in separate kernel invocation, Pensieve handles both the prefill phase and generation phase in a unified way. In other words, the Pensieve scheduler forms a batch of requests regardless of which phase they are in. In the same batch, some could be in their generation phase while others are in their prefill phase. Such unified batching is made possible by Pensieve's multi-token attention kernel design (§4.4.1).

Pensieve's iteration-level scheduler is "clocked" for action by the completion of a generation step. In particular, after each iteration of token generation, the worker returns any finished request that has either emitted the end-of-sentence token or reached maximum decoding length. The scheduler finds the next request in its wait queue to join the batch, if there is room (i.e. the total number of tokens in the batch is fewer than some pre-configured threshold). We use the simple first-come-first-serve scheduling policy when choosing which new requests to join the batch.

For unified batch formation, the scheduler concatenates the tokens to be processed from all requests. For each new request joining the batch, its tokens include those corresponding to the request's new user prompt. For each existing request in the batch, its token includes the one generated by

the last generation step. By combining the prefill phase together with the generation phase, we avoid running separate small kernels and can thus improve GPU utilization.

4.3 KV cache management

Traditionally, the KV cache in GPU only serves as a computation workspace. In Pensieve, the GPU KV cache also serves as storage space to cache the KV-tokens of recently completed requests of active conversations. Pensieve adopts a two-tier hierarchical caching strategy and uses the much larger CPU memory as the second-tier cache space. The scheduler determines the caching plan and instructs the worker to perform the actual data movements between GPU and CPU.

The scheduler tracks the amount of free GPU KV cache slots left. Before handing off a batch of requests to the worker, the scheduler tries to ensure that any new request's past KV-tokens will reside in the GPU and there is sufficient GPU memory for the execution. Specifically, if a request in the batch has any KV-tokens that have been swapped out to the CPU memory or dropped, the scheduler determines how many additional GPU KV cache slots are needed to swap in or re-calculate those absent KV-tokens. If there is sufficient space, the scheduler instructs the worker to perform the necessary allocation and to start swapping those KV-tokens from the CPU as part of the batch's cache plan.

4.3.1 Eviction Policy. Pensieve performs fine-grained token-level cache eviction and dropping. We design the eviction policy to express two kinds of preferences: 1) it preferentially evicts from older conversations, aka those that have been inactive for a longer period of time. This is based on the same LRU assumption that the least recently active conversation will not see activity for a longer time. 2) it preferentially evicts tokens from the leading end of a conversation's history context. This is based on the observation, previously shown in Figure 4, that leading tokens of a conversation are cheaper to recompute than trailing ones. Below, we describe how our policy evicts according to these preferences.

Eviction granularity. In order to reduce the overhead caused by frequent eviction decision-making and moving small amounts of memory over the PCIe bus, we group KV-tokens into chunks and make eviction decisions at the granularity of chunks. The chunk size is configurable. In our experiments, we find that setting the chunk size to 32 tokens works well.

The retention value of a chunk. In order to combine both the LRU preference and the evicting from the front preference, we calculate a score for each chunk to capture its retention value. Specifically, the retention value of a chunk is $V = \frac{Cost(s,l)}{T}$, where $Cost(s,l)$ represents the cost of recomputing a chunk of size s with a context of size l, and the denominator T is the amount of time since the conversation was last active. Pensieve evicts chunks according to

the ascending order of their retention values so that chunks with lower recomputation cost and/or those belonging to conversations with longer inactive periods are preferentially evicted.

Estimating the recomputation cost. In order to calculate the retention value of a chunk, we need to estimate its recomputation cost. In particular, we view the cost to recompute the embedding of a chunk as the sum of recomputing the LLM model's attention operation and the rest of the non-attention operation: $Cost(s,l) = Cost_{attention}(s,l) + Cost_{other}(s)$, where s is the chunk size and l is the size of the context to which the chunk "attends" for the attention operation. The cost of non-attention computation cost ($Cost_{other}(s)$) consists of linear layers, layer normalization, non-linear activation, etc., and therefore is independent of the context size. On the contrary, attention requires accessing and performing computation with all l context tokens. Since the eviction decisions are made for fixed-size chunks of 32 tokens, we can simplify the cost function to become $Cost(l) = Cost_{attention}(l) + c$ where $Cost_{attention}(l)$ is the cost of performing attention operation for a chunk of 32 tokens with context length l, and c is a constant capturing the cost of non-attention computation. We perform offline profiling to estimate c as well as $Cost_{attention}(l)$ with varying context sizes. Since it's not feasible to profile all possible context sizes, we profile context sizes that are powers of 2 and use the measured values to interpolate the cost for other context sizes.

4.3.2 Ahead-of-the-time swapping. Since Pensieve tries to preserve KV-tokens in the GPU for reuse by a later request in the same conversation, the scheduler does not immediately release a request's GPU cache slots as soon as it finishes, unlike existing systems [27]. Instead of waiting until the GPU cache has run out, the scheduler asks the worker to copy (aka swap out) selected KV-tokens to the CPU if less than a threshold (e.g. 25%) of the GPU cache slots are available. The corresponding GPU memory is reclaimed in a lazy manner and is not immediately released until the scheduler later decides to allocate the same slots to another conversation.

When the CPU cache runs out of space, the same eviction policy (§4.3.1) is used to decide which KV-tokens to drop. Performing ahead-of-the-time swapping allows the scheduler to overlap cache eviction with GPU computation, thus fully hiding the latency of swapping.

4.3.3 Pipelined KV cache recovery. The scheduler does not wait for a request's KV-tokens to be fully swapped in from the CPU before handing it off to the worker for execution. Rather, we follow the pipelined approach [4] to overlap computation with data transfer. Specifically, we exploit the fact that an LLM model has many layers and each layer's KV-token is only used in this layer's self-attention calculation. Instead of waiting for all layers to finish data transfer before

starting the execution, we initiate the transfer layer by layer and start model computation at the same time. The worker uses GPU events to preserve data dependency: it only starts a layer's self-attention kernel once that layer's KV-tokens have been fully copied to the GPU. Pipelining transfer with computation allows us to hide the swap-in latency.

4.3.4 Handling dropped tokens.

If a request has some of its KV-tokens dropped due to CPU memory pressure, we resort to recomputation to handle such dropped tokens. Figure 5 shows the whereabouts of a typical new request's KV-tokens. As we always evict cached tokens from the leading end of a conversation, the GPU cache generally holds the request's latest tokens and the CPU cache holds the middle, while the earlier ones may have been dropped.

The scheduler swaps in those tokens cached in the CPU. For dropped tokens, the scheduler will fetch their corresponding raw text tokens from the conversation history saved in a persistent store (Figure 7). These retrieved raw tokens will be merged into (i.e. prepended to) the new request's prompt and become part of the batch's input tokens, as shown in Figure 8 (step a). In the prefill phase, the embedding of dropped tokens and new prompt tokens are concatenated together as they are processed by successive model layers. At each Transformer layer, the Query, Key, and Value tensors are computed (step b). Key and Value are stored in KV cache, and Pensieve maintains the KV locations for the entire conversation context including previously cached tokens (step c), which can then be used to perform attention.

However, as discussed in §3.2, the challenge that comes with dropping leading tokens is that tokens in query tensor correspond to two disconnected ranges in the context, while all existing attention kernels assume that the Query tensor region is consecutive. To address this, we treat these two ranges as two sub-requests that happen to share portions of the underlying context. Each row in Figure 8 (step d) represents the Query tensor and its corresponding KV context locations of each sub-request of the original request. As our multi-token attention GPU kernel design (§4.4) can support Query tensors of variable lengths for different requests in the batch and also accept non-contiguous KV-token locations, Pensieve only needs to update auxiliary data structures and no memory copy is incurred when processing the sub-requests.

4.3.5 Suspending requests during generation.

Despite ahead-of-time eviction, the scheduler might still encounter scenarios when the generation phase runs out of GPU cache since a request's decoding length is not known a priori. In this situation, the scheduler suspends some requests' execution by taking them out of the current batch, swaps out their corresponding KV-tokens to the CPU, and puts them back in the waiting queue. It chooses which request to suspend according to the descending order of their arrival time. As suspension causes increased latency (due to waiting for the

swap-out), we try to avoid it by conservatively reserving 10% of GPU cache slots for the execution of existing requests that are in the generation phase. In other words, the scheduler stops adding new requests into the running batch unless there are more than 10% free GPU cache slots.

4.4 Multi-token attention for non-contiguous cache

How to combine existing KV-tokens in the GPU cache with those just swapped in from the CPU? The most naive solution is to allocate a contiguous memory region in the GPU to hold them both. However, doing so would incur expensive memory copying, since KV-tokens are large. A more promising solution is to allocate separate space only for those swapped-in KV-tokens and to design an attention kernel implementation that can handle non-contiguous memory in its KV cache.

vLLM's PagedAttention kernel can handle non-contiguous KV cache in the generation phase [27]. However, for the prefill phase, it still uses existing kernels which require all KV-tokens to reside in contiguous memory. We cannot use PagedAttention because it assumes each request in a batch has exactly one input token, which is the case for generation but not prefill. Thus, we refer to PagedAttention as a single-token attention kernel because it computes the attention scores between a single input token's query representation (Q) and the KV cache. We need to build a multi-token attention kernel that supports performing attention between the query representations (Q) of multiple input tokens per request and the KV cache over non-contiguous memory.

Our new kernel is similar to that of vLLM to the extent that both kernels need to support loading KV cache from non-contiguous GPU global memory into on-chip shared memory. Their main difference is illustrated in Figure 9. As vLLM's kernel performs attention between a request's single input token and the existing KV-tokens, its underlying computation can be described as two matrix-vector multiplication operations[1], as shown in Figure 9. In contrast, our kernel handles multiple input tokens for each request, computing attention scores between all pairs of input tokens and the conversation's existing KV-tokens. Therefore, its underlying computation can be described as two matrix-matrix multiplication operations, and the batched version of our kernel performs batched matrix-matrix multiplications. Like vLLM, we fuse the two multiplication operations according to [10]. Because of the additional dimension in the Q tensor, our kernel has more parallelization and tiling opportunities on the GPU. However, care must be taken to handle the new challenge that Q is a ragged-sized tensor as different requests in the batch have different numbers of input tokens.

When multiple input tokens of a request are handled together in a kernel, we need to apply causal masking so that

---

[1]The batched version of PagedAttention performs batched matrix-vector multiplications.
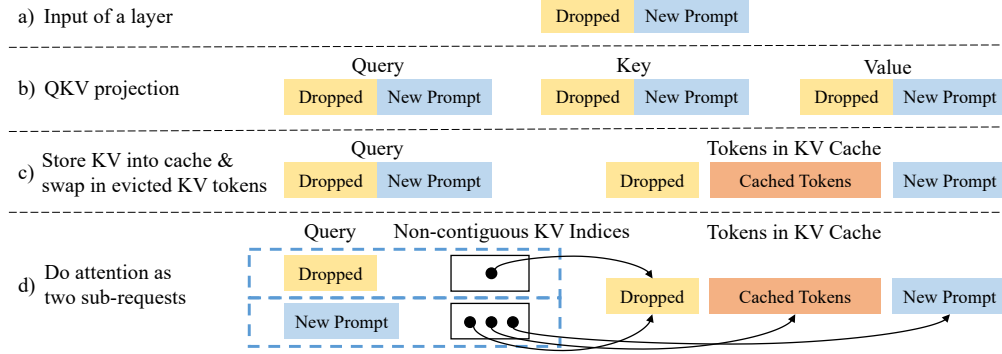
Figure 8. How Pensieve handles dropped tokens. (a) concatenate tokens of the dropped context and new prompt as the input. (b) Calculate projections Q, K, V (c) Update KV cache using newly calculated K, V; swap in KV-tokens previously evicted to the CPU. (d) Do attention for a batch of two sub-requests: one for the dropped context to attend to itself, and the other for the new prompt to attend to the entire context.
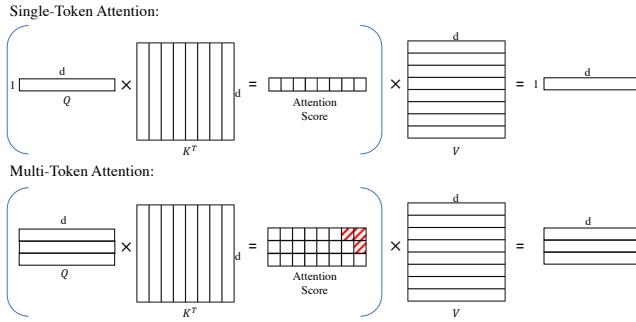

Figure 9. Single-token attention vs multi-token attention, softmax and rescaling omitted. d is the dimension of the embedding vector. The red shaded area in the multi-token attention score represents causal masking. K and V are logically contiguous, but physically placed at random locations when paged memory management is used.

an earlier input token does not "attend" to later tokens. This requires setting a corresponding upper triangular part of the attention score matrix to 0, as shown by the red shaded area in Figure 9. Causal masking is not needed for the single-token attention kernel since it only processes one input token. We fuse the causal masking operation inside the multi-token attention kernel to avoid materializing the intermediate attention score matrix [10].

Matrix-matrix multiplication kernels are much more complex than those for matrix-vector multiplication because they typically use sophisticated algorithms to extract additional data reuse opportunities with GPU's on-chip shared memory and to leverage GPU's tensor core primitives. Therefore, instead of trying to extend vLLM's PagedAttention kernel for multi-token attention, we base our implementation on an existing multi-token attention kernel from PyTorch and extend it to handle non-contiguous KV cache. Our kernel uses the high-performance thread-block level matrix-matrix multiplication provided by NVIDIA Cutlass template library [33].

4.4.1 Unifying the prefill and generation phase.
As discussed in §4.2, in Pensieve's batch formation, new requests in their prefill phase can be grouped with existing requests in their generation phase. This is enabled by our multi-token attention kernel because single-token attention performed in the generation phase can be treated as a special case of multi-token attention with query size equal to 1.

More concretely, Pensieve's scheduler concatenates the input tokens to be processed from all requests, regardless of whether they correspond to prompts of new requests or the last step's output token from existing requests. Some auxiliary data structure is maintained to keep track of each request's corresponding region. During execution by the worker, these batched input tokens are fed through linear layers to generate each token's QKV representations. Newly generated KV-tokens are stored in the allocated slots in the GPU's KV cache. Then the worker applies our multi-token attention kernel to produce the output tokens for all requests.

4.4.2 Multi-GPU support.
For large LLM models that do not fit into a single GPU, we adopted Tensor Parallelization, as implemented in Megatron-LM [45], to partition large models across multiple GPUs. The KV cache in Pensieve is also partitioned accordingly. Each model partition is managed by a worker process which has its own GPU and CPU cache partition to store the attention state partition assigned to this worker. Since the model partitioning occurs along the feature dimension of KV cache, it does not affect the eviction policy that decides which token to migrate or drop. Therefore, each worker follows the same migration plan to swap attention states between GPU and CPU memory.

5 Implementation

We have implemented our prototype serving system Pensieve with ~7K lines of C++/CUDA code. Pensieve manages KV cache and auxiliary data structure needed by multi-token attention on the GPU, but relies on PyTorch (v2.0.0, CUDA 11.8) C++ front-end APIs to execute GPU operators in Large

Model	OPT-13B	OPT-66B	Llama 2-13B	Llama 2-70B
# layer	40	64	40	80
# hidden	5120	9216	5120	8192
# head	40	72	40	64
# KV head	40	72	10*	8
Head size	128	128	128	128
# GPU	1	4	1	4

Table 1. Hyper-paramaters for OPT and Llama 2 models.

	ShareGPT	UltraChat
# conversations	48,159	1,468,352
Mean # of turns	5.56	3.86
Mean request input length	37.77	51.78
Mean request output length	204.58	257.81

Table 2. Dataset statistics

Language Models. Based on PyTorch's fused memory efficient attention, we develop our own fused multi-token attention kernel using NVIDIA Cutlass library to support performing attention with KV-tokens that reside in non-contiguous memory.

Optimization: Prioritize data retrieval over eviction. Although PCIe allows full-duplex bidirectional data transfer, in practice, we found that when CPU-to-GPU data transfer is done concurrently with GPU-to-CPU data transfer, there is a significant throughput drop (18-20%) in both directions. Similar issues have been reported[2]. Since Pensieve performs KV-token swap-out ahead of time, there is no urgency to finish the transfer right away. To prevent eviction from slowing down the swapping in of past KV-token, we set up a waiting mechanism. In particular, if a worker has any ongoing swap-in task, it waits to perform GPU-to-CPU copy until the swap-in task is done. Although this conservative approach does not fully utilize the duplex PCIe bandwidth, we find that this optimization performs well and we never run into the situation that the GPU-to-CPU copying can't catch up.

6 Evaluation

6.1 Experimental Setup

System Environment. We evaluate Pensieve on Azure NC A100 v4 series, which are equipped with up to 4 A100-80GB GPUs, a 24-core AMD EPYC 7003 processor and 220 GB CPU memory per GPU. For each system evaluated, we configure it to allocate 40 GB memory on each GPU for KV cache for a fair comparison.

Models. We use two open-source models: OPT [55] and Llama 2 [47]. OPT has an almost identical model architecture to GPT-3 [5] while Llama 2 is a more recent model that employs more advanced model features like rotary embedding, RMS Layernorm [54], SiLU, etc. Notably, Llama 2 follows the trend of adopting Grouped-Query Attention (GQA) [2] which divides query heads into groups so that only one KV head is used within each group. GQA significantly reduces the memory consumption of KV-tokens, allowing Pensieve to store more past KV-tokens.

We evaluate two different sizes for each model: a small one on a single GPU, and a large one partitioned onto 4 GPUs using Tensor Parallelization as done in Megatron-LM (§4.4.2). Detailed model hyper-parameters can be found in Table 1. By default, Llama 2 only uses GQA for models with over 70 billion parameters. To demonstrate Pensieve's effectiveness when used with GQA, we changed the number of KV heads of Llama 2-13B from 40 to 10. In all experiments, the 16-bit half-precision float format is used for both model parameters and intermediate hidden representations.

Dataset. We evaluate Pensieve on two multi-turn conversation datasets: ShareGPT and UltraChat. ShareGPT [42] is a real-world dataset containing user-shared ChatGPT conversations. UltraChat [14] is a recent large-scale synthetic dataset for multi-turn dialogue: it uses separate LLMs to simulate the interaction between a user and the chatbot assistant. Table 2 shows the statistics of both datasets. In our experiments, we limited the maximum context size to 16384 tokens and dropped 0.57% of the conversations in ShareGPT dataset that exceed this limit.

Workload. Since the datasets do not provide timestamps for each user request, we simulate a request's arrival time by sampling from a Poisson distribution under different request rates. We maintain the causal dependency for requests belonging to the same conversation: a new user prompt is only sent to the system after the response to the conversation's previous request has been received. Additionally, we also simulate user think time, aka the time taken for users to generate the next conversation turn, by sampling from an exponential distribution with varying mean.

Baselines. We evaluate Pensieve against two state-of-the-art serving systems: vLLM [27] (v0.2.0) and TensorRT-LLM [36] (v0.12.0). Both of them use a stateless serving API. For each request, the new user prompt is appended to the history and then processed as an input request. vLLM uses PyTorch as its execution backend. By contrast, TensorRT-LLM compiles and optimizes the model using graph rewriting optimizations such as operator fusion, and executes the optimized model using the TensorRT Runtime. Since vLLM is an execution engine without a serving loop that drives the engine to process incoming requests, we implement such a driver that adds newly arrived requests into vLLM job queue and invokes the engine execution until all requests are fully processed.

---

[2]https://forums.developer.nvidia.com/t/data-transfers-are-slower-when-overlapped-than-when-running-sequentially/187542

(a) OPT-13B, ShareGPT

(b) Llama 2-13B, ShareGPT

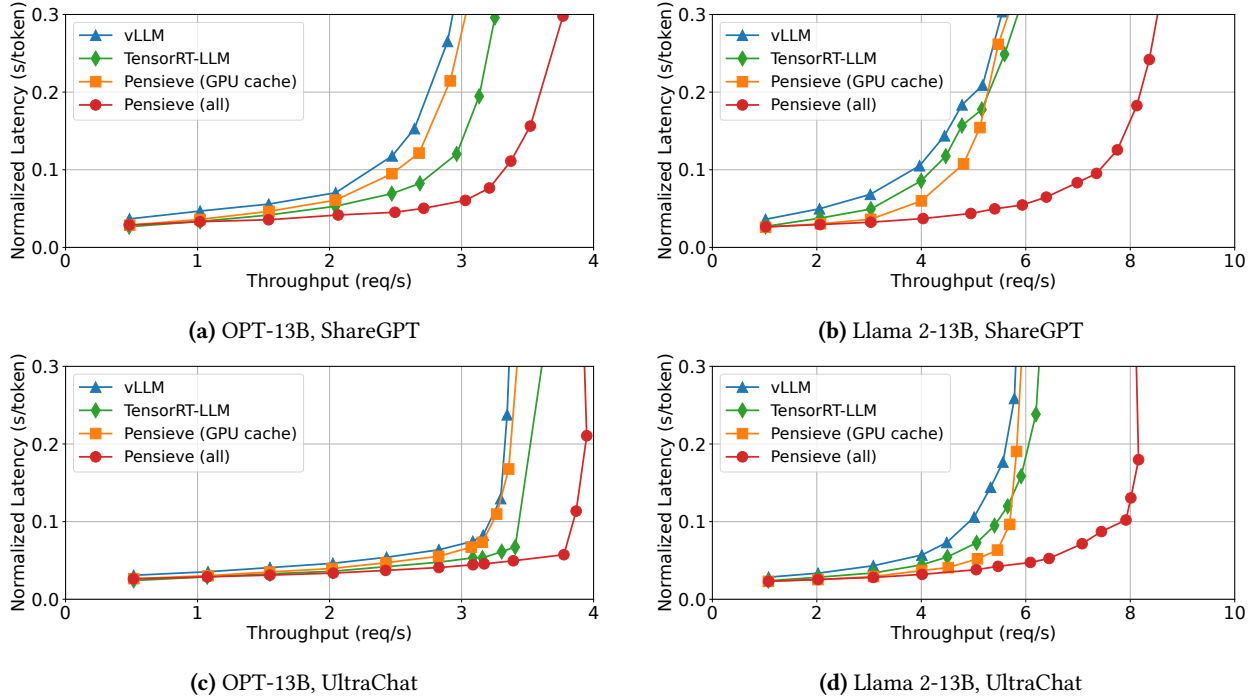**(c)** OPT-13B, UltraChat

(d) Llama 2-13B, UltraChat

Figure 10. LLM serving performance on 1 GPU.

We also experiment with a variant of Pensieve called Pensieve (GPU cache) that simply drops evicted tokens from the GPU instead of swapping them out to the CPU. This variant is used to examine the effectiveness of CPU caching in Pensieve.

Performance Metric. Pensieve is designed to optimize both peak throughput and latency of serving LLM in conversational scenarios. Following prior work [27, 52], we measure the achieved serving throughput and 90-percentile normalized latency, which is calculated as the end-to-end request serving latency divided by the number of output tokens.

6.2 End-to-End Serving Performance

Figure 10 shows the normalized latency vs throughput for OPT-13B and Llama 2-13B served with a single A100-80GB GPU. The normalized latency is calculated as the mean of each request's end-to-end latency divided by its output length, as done in [27, 52]. On the ShareGPT dataset shown in Figure 10(a) and (b), Pensieve's throughput is 1.36× of vLLM and 1.14× of TensorRT-LLM for serving OPT-13B at 120 ms per token latency, and 1.70× of vLLM and 1.58× of TensorRT-LLM for serving Llama 2-13B at 180 ms per token latency. On the UltraChat dataset shown in Figure 10(c) and (d), Pensieve's throughput is 1.17× and 1.14× of vLLM and TensorRT-LLM respectively for serving OPT-13B at 120 ms per token latency, 1.58× and 1.47× for serving Llama 2-13B at 100 ms per token latency. Pensieve has more throughput gains on ShareGPT than UltraChat because the real world dataset ShareGPT has

more conversation turns (Table 2), which makes saving past KV-tokens more beneficial.

In Figure 10, TensorRT-LLM outperforms vLLM consistently. It even outperforms Pensieve (GPU cache) in Figure 10(a) and (c). This is because both Pensieve and vLLM use PyTorch API to execute the model while TensorRT-LLM performs optimizations like graph rewriting and operator fusion during an offline compiling stage, and then executes the optimized model using the TensorRT Runtime. However, by avoiding recomputing past KV-tokens for continuing conversations, Pensieve achieves significant throughput gain over TensorRT-LLM.

Pensieve's performance gain is more significant for Llama 2-13B than for OPT-13B because our version of Llama 2-13B uses Grouped-Query Attention with group size 4 (i.e. every four query heads share a single key-value head). Consequently, the amount of memory required to store past KV-tokens is reduced by 4×, thus allowing Pensieve to store more past KV-tokens and better avoid recomputation.

When Pensieve is used without CPU cache, i.e. Pensieve (GPU cache), it may still reduces latency because baseline systems always recompute past KV-tokens for requests from a returning conversation, and thus, the number of tokens processed in the prefill phase is on average larger. But under relatively higher request rates, GPU cache is quickly exhausted, and Pensieve (GPU cache) also resorts to recomputing past KV-tokens from scratch.
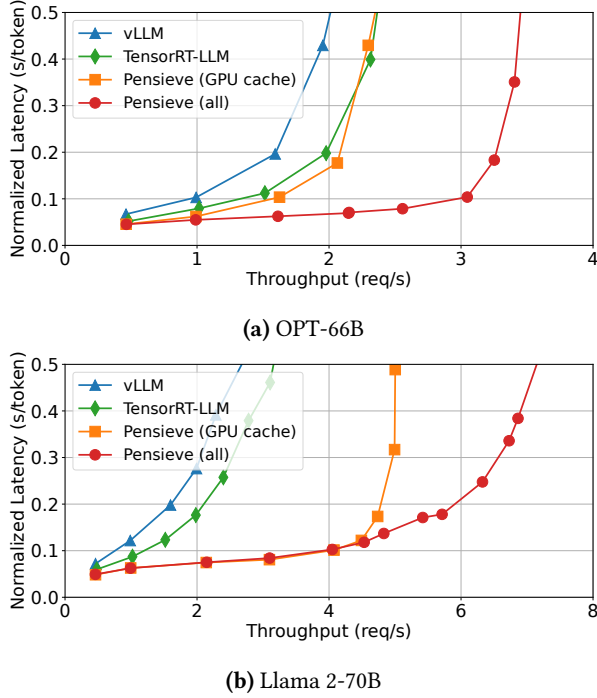
(a) OPT-66B

(b) Llama 2-70B

Figure 11. LLM serving performance on 4 GPUs for ShareGPT.

6.3 Multi-GPU Serving Performance

Figure 11 shows the performance of Pensieve for larger models, OPT-66B and Llama 2-70B, when run on four GPUs using the ShareGPT dataset. Larger models amplify Pensieve's advantage over the baselines because the amount of computation grows faster than the memory usage of KV-tokens. For example, from OPT-13B to OPT-66B, the model parameter size and computation amount increase by more than 5×, while the KV cache size (# layer × # hidden × 2) only increases by 2.88× (Table 1). Since the number of GPUs and CPU memory are usually scaled linearly with the model size, Pensieve can store more past KV-tokens in its KV cache.

Pensieve achieves 2.04× the throughput of vLLM and 1.64× of TensorRT-LLM for serving OPT-66B at 200 ms per token latency, and 3.0× of vLLM and 2.47× of TensorRT-LLM for serving Llama 2-70B at 400 ms per token latency. The improvement is more significant on Llama 2-70B because it uses Grouped-Query Attention with group size 8, which reduces the memory requirement for past KV-tokens by 8×. This much-reduced memory consumption in KV-tokens also significantly benefits the throughput of Pensieve (GPU cache), as shown in Figure 11(b).

6.4 Performance of Multi-token attention Kernel

Figure 12 shows the performance of Pensieve's multi-token attention kernel compared to alternative implementations. In this microbenchmark, we measure the latency of the attention operator for a batch of 32 requests each with a prompt
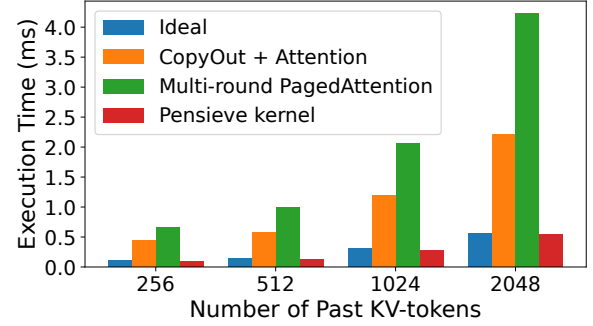

Figure 12. Execution time of Pensieve's multi-token attention kernel over non-contiguous context memory with different context sizes. Measured with batch size 32 and query size 8.

of 8 query tokens and different numbers of past KV-tokens stored in non-contiguous GPU memory. As described in §3.1, existing attention kernels are not directly applicable. We compare against two straw-man implementations: (1) "CopyOut+Attention" allocates additional contiguous GPU memory to copy past KV-tokens into and then invokes an existing fused attention kernel (orange bar), and 2) "Multi-round PagedAttention" invokes multiple rounds of vLLM's single-query PagedAttention to process one token from the prompt at a time (green bar). We also show the performance of the ideal situation which assumes that past KV-tokens are stored in contiguous memory space (blue bar). Figure 12 shows that both straw-man solutions add significant overhead compared to the ideal performance. Copying to contiguous memory incurs cost proportional to the number of past KV-tokens. Applying multiple rounds of PagedAttention, on the other hand, gives up parallelization opportunities on prompt tokens, resulting in execution time linear to the number of tokens in the prompt. Pensieve's kernel matches the ideal baseline. In fact, it has slightly better performance because we offload auxiliary data computing (like calculating the cumulative sum for the sequence length of a batch) to the CPU. Since each transformer layer in the model shares the same caching plan, these auxiliary data can be reused by all layers.

6.5 Effect of Unified Scheduling

We evaluate whether using a unified scheduler for both prefill and generation phases is beneficial for performance. Figure 13 shows the performance of Pensieve with and without unified scheduling for Llama 2-13B on the ShareGPT dataset. Compared to processing prefill and generation phases separately, unifying them into a single execution step avoids
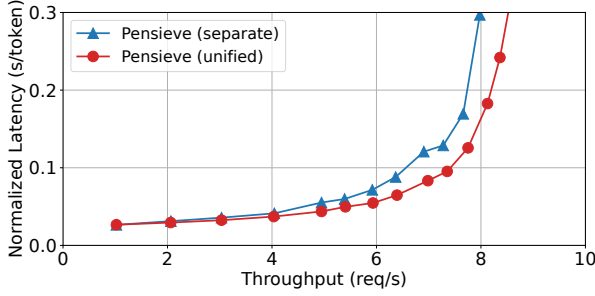
Figure 13. Performance of serving Llama 2-13B with and without unified scheduling on ShareGPT dataset.
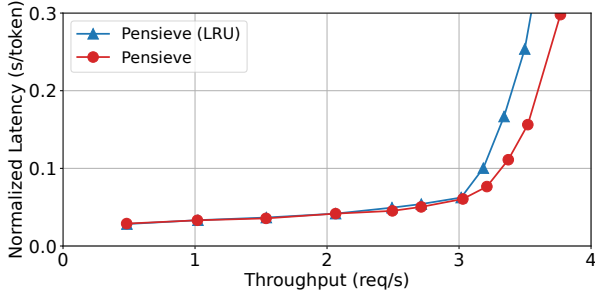

Figure 14. Performance of serving OPT-13B with different eviction policies on ShareGPT dataset.

having to execute the prefill phase with a small number of requests. As a result, Pensieve with unified scheduling achieves better throughput and latency.

6.6 Effect of the Eviction Policy

We compare Pensieve's caching policy against the classic LRU policy. We use OPT-13B as the workload for this evaluation. Figure 14 shows that both policies exhibit similar performance until the workload approaches 3 requests per second, beyond which Pensieve's eviction policy outperforms LRU. After analyzing the execution traces, we find that both policies have less than 80% cache hit rate, however, Pensieve's policy has up to 4.4 percentage points higher CPU cache hit rate than LRU. On average, Pensieve's policy reduces the number of recomputed KV-tokens by up to 14.6%.

6.7 Impact of User Think Time

Our experiments so far use the average user think time of 60 seconds. Figure 15 evaluates the impact of different average user think times on the performance of Pensieve using Llama 2-13B. Additionally, we also show vLLM with 600-second think time as a comparison point. As seen in Figure 15, the throughput of Pensieve decreases as the average user think time increases, causing past KV-tokens to drop from the cache at a higher rate. Even as user think time increases to 600 seconds, Pensieve still achieves better latency and
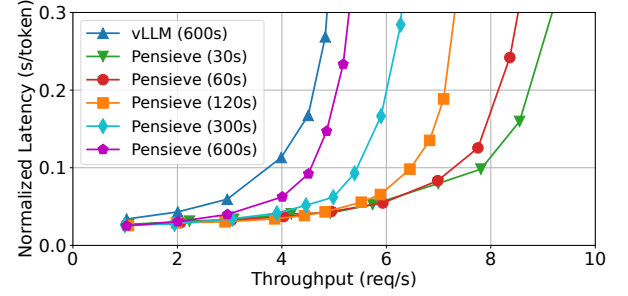

Figure 15. Performance of serving Llama 2-13B with different average user reaction time on ShareGPT dataset.

throughput compared to vLLM, although the performance gap becomes smaller than that for smaller think times.

7 Related Works

LLM inference and serving systems. Many systems have been recently developed to serve language models with better performance including vLLM [27], ORCA [52], TensorRT-LLM [36], FasterTransformer [35], LightSeq [49], DeepSpeed [3] and [38]. These systems investigate different performance improvement opportunities than Pensieve. The wide range of proposed techniques include incremental decoding [35], iteration-level batching [17, 52], supporting paged KV cache over non-contiguous GPU memory [27, 36], finding an efficient multi-device partitioning plan [38], better load balancing between prefill and generation to reduce pipeline parallelism bubbles [1, 20], kernel fusion [10, 11, 39], speculative decoding [6, 28], and quantization [13, 18, 31, 51].

ServerlessLLM [15] proposes techniques to do fast model loading and live migration in order to serve requests in a serverless setup. DistServe [57] advocates performing the prefill and generation phase of the same request on separate GPUs to better satisfy SLA. By contrast, Pensieve performs unified batching of both phases because we aim to optimize throughput instead of latency SLA.

Systems that cache LLM attention states. Several systems (Table 3), developed concurrently with Pensieve, also cache attention states in LLM serving to reduce computation.

PromptCache [19], SGLang [56], RAGCache [25] and CachedAttention [16] all cache attention states of shared text segments across different requests. Among these systems, CachedAttention targets multi-turn conversations, RAGCache targets retrieval-augmented generation (RAG) while others aim to be general. PromptCache requires users to define a schema a priori to explicitly specify any reuse opportunities. This requirement is too restrictive since static schemas cannot specify those dynamic prefixes to be cached in multi-turn conversations. SGLang and RAGCache employ a tree-based cache in order to enable prefix sharing across different users or different RAG document prefixes. By contrast,

	PromptCache [19]	SGLang [56]	CachedAttention [16]	RAGCache [25]	CacheGen [29]	Pensieve
Cache shared across users	Yes	Yes	No	Yes	Yes	No
Cache storage	CPU	GPU	CPU & Disk	CPU & GPU	Network	CPU & GPU
Eviction granularity	N/A	requests	entire conversation history	documents	N/A	tokens
Eviction location preference	N/A	trailing	N/A	trailing	N/A	leading

Table 3. Comparison of Pensieve with other caching systems for LLM attention states.

Pensieve and CachedAttention are designed specifically for multi-turn conversation scenarios where different users are unlikely to share conversational content[3]. In this setting, there is no need for the extra complexity of a tree-based cache structure. Instead, Pensieve and CachedAttention both focus on managing a dynamic cache for ongoing conservation histories which grow in size over time.

As Table 3 shows, the important difference between Pensieve and others (SGLang [56], RAGCache [25] and CachedAttention [16]) lies in Pensieve's eviction strategy which evicts at the finer-granularity of groups of tokens at the leading end to reduce the cost of recomputation. By contrast, CachedAttention evicts at the granularity of an entire conversation and does not recompute truncated conversation context, while SGLang and RAGCache evict at the granularity of a tree node containing an entire user request or document. SGLang and RAGCache also preferentially evict from the bottom of the tree (aka the trailing end of a prefix) as opposed to the leading end as done in Pensieve. Pensieve's eviction strategy introduces challenges in supporting computation in non-sequential regions of request context (Figure 5), which is addressed in Pensieve's multi-token attention kernel (§4.3.4).

Another recent system, CacheGen [29], addresses the problem of reducing the context-loading delay for long contexts stored in remote network storage. CacheGen proposes an adaptive compression scheme to accelerate the transfer of the KV cache over a bandwidth-limited and bursty network. As Pensieve only keeps the KV cache in local GPU and CPU memory, CacheGen's techniques are orthogonal to those of Pensieve. For conversational chatbots, Pensieve using local cache can already bring significant performance gains.

Non-LLM specific DNN serving systems. Systems like TensorFlow Serving [37], Clipper [9], NVIDIA Triton Inference Server [34], Nexus [43], and InferLine [8] serve as scheduling components of a general-purpose DNN serving system. They are mostly model-agnostic and execution backend-agnostic and apply general system techniques like batching, caching, and software pipelining to serve DNN models. Some are also in charge of properly provisioning compute resources to improve overall cluster efficiency. A few existing works target model-less serving and provide inference as a service: they automatically select models to meet the accuracy and latency requirements of a given user task. For example, INFaaS [41] and Tabi [50] serve requests with a small model and only re-route to a larger model when the output confidence score is low.

Techniques addressing the GPU memory limit. These include GPU-CPU swapping, recomputation, and unified memory. Most of the systems described below are not specifically targeted for LLM serving.

GPU-CPU Swapping: SwapAdvisor [22] swaps weight and activation tensors for DNN training. It uses the dataflow graph to determine an optimal plan that involves operator execution order, memory allocation, and swapping. Zero-Offload [40] offloads optimizer state and gradients to the CPU during LLM training with a single GPU. DeepSpeed-ZeroInference [3] and FlexGen [44] use offloading to serve LLM with a weak GPU. DeepSpeed-ZeroInference offloads entire model weights to CPU or NVMe memory. FlexGen offloads currently unused model weights, activation, or KV cache to CPU memory or disk. As a result of frequent data movement and high disk latency, these two systems require a large batch size to hide the offloading latency. Therefore, they mainly target latency-insensitive applications. Neither system persists the KV cache across requests.

Recomputation: For DNN training, recomputing activation on the backward pass [7] is a popular technique used to reduce memory footprint. The decision for which tensors to compute is done at the granularity of layers [7], a group of operators within a layer (aka modules) [26], or individual operators [23]. Unlike these works, Pensieve's fine-grained recomputation is done at the token(chunk)-level, to exploit the insight that earlier tokens in a sequence incur less recomputation cost due to the causal nature of attention.

Unified Memory and Direct Host Access: Swapping between CPU and GPU memory can also be achieved transparently through Unified Memory [32], which automatically triggers memory page migration between CPU and GPU using a page fault mechanism. The Direct Host Access feature allows a GPU kernel to directly read from CPU memory, thereby allowing a subset of the threads to execute as soon as their data is ready without waiting for the entire transfer to finish. It has been used to enable better overlap of data transfer and kernel computation when swapping in a model from CPU [30] or accessing large graph neural network features on the CPU [24]. For our implementation of Pensieve, we choose not to use Unified Memory nor Direct Host Access because these mechanisms trigger memory transfer only

---

[3]If the chatbot uses a common system prompt across users, it can be handled by explicitly designating the system prompt state as reusable.

when the data is accessed by a GPU kernel and we want to manage data movement explicitly to prefetch the KV cache. Furthermore, for Direct Host Access, since the copied data is not explicitly stored in GPU memory, it has to be loaded from the CPU again if it is repeatedly used. In our workload, the context cache is not only used to prefill the prompt tokens but also in every generation step, which makes Direct Host Access significantly less efficient.

8 Conclusion

When serving Large Language Models for multi-turn conversations, a major inefficiency is due to the recomputation of a conversation's cumulative history context. We develop Pensieve, a stateful LLM serving system that preserves history embeddings in a multi-tier GPU-CPU cache. It uses a new GPU attention kernel to perform attention between requests' new multi-token input and their saved context stored in non-contiguous GPU memory. Experiments show that Pensieve achieves 1.14-1.70× the throughput of baseline systems for small models (OPT-13B, Llama 2-13B) and 1.64-3.0× for larger models (OPT-66B, Llama 2-70B).

Acknowledgments

References

[1] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. 2023. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369* (2023).

[2] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245* (2023).

[3] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. 2022. DeepSpeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[4] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. 2020. PipeSwitch: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 499–514.

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[6] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. 2023. Accelerating

[7] Large Language Model Decoding with Speculative Sampling. In *arXiv:2302.01318*.

[7] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).

[8] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. 2020. InferLine: latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 477–491.

[9] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 613–627.

[10] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing Systems*.

[11] Tri Dao, Daniel Haziza, Francisco Massa, and Grigory Sizov. 2023. Flash-Decoding for long-context inference. https://pytorch.org/blog/flash-decoding/.

[12] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314* (2023).

[13] Tim Dettmers and Luke Zettlemoyer. 2023. The case for 4-bit precision: k-bit inference scaling laws. In *International Conference on Machine Learning*. PMLR, 7750–7774.

[14] Ning Ding, Yulin Chen, Bokai Xu, Yujia Qin, Zhi Zheng, Shengding Hu, Zhiyuan Liu, Maosong Sun, and Bowen Zhou. 2023. Enhancing Chat Language Models by Scaling High-quality Instructional Conversations. *arXiv preprint arXiv:2305.14233* (2023).

[15] Yao Fu, Leyang Xue, Yeqi Huang, Andrei Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. ServerlessLLM: Locality Enhanced Serverless Inference for Large Language Models. In *OSDI*.

[16] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. 2024. Cost-Efficient Large Language Model Serving for Multi-turn Conversations with CachedAttention. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 111–126.

[17] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. 2018. Low latency rnn inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference*. 1–15.

[18] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. 2022. A survey of quantization methods for efficient neural network inference. In *Low-Power Computer Vision*. Chapman and Hall/CRC, 291–326.

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

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

[21] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, and Weizhu Chen. 2021. LoRA: Low-Rank Adaptation of Large Language Models. (2021). arXiv:2106.09685 https://arxiv.org/abs/2106.09685

[22] Chien-Chin Huang, Gu Jin, and Jinyang Li. 2020. SwapAdvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1341–1355.

[23] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. 2020. Checkmate: Breaking the memory wall with optimal tensor rematerialization. *Proceedings of Machine Learning and Systems* 2 (2020), 497–511.

[24] Jinwoo Jeong, Seungsu Baek, and Jeongseob Ahn. 2023. Fast and Efficient Model Serving Using Multi-GPUs with Direct-Host-Access. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 249–265.

[25] Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Xin Liu, Xuanzhe Liu, and Xin Jin. 2024. RAGCache: Efficient Knowledge Caching for Retrieval-Augmented Generation. In *arXiv:2404.12457*.

[26] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2023. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems* 5 (2023).

[27] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

[28] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *ICML*.

[29] Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, et al. 2024. CacheGen: KV Cache Compression and Streaming for Fast Large Language Model Serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 38–56.

[30] Seung Won Min, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei Hwu. 2021. Large graph convolutional network training with gpu-oriented data communication architecture. *arXiv preprint arXiv:2103.03330* (2021).

[31] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart Van Baalen, and Tijmen Blankevoort. 2021. A white paper on neural network quantization. *arXiv preprint arXiv:2106.08295* (2021).

[32] NVIDIA. 2017. CUDA Unified Memory. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/.

[33] NVIDIA. 2017. CUTLASS. https://github.com/NVIDIA/cutlass.

[34] NVIDIA. 2018. Triton Inference Server. https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/index.html.

[35] NVIDIA. 2021. FasterTransformer. https://github.com/NVIDIA/FasterTransformer.

[36] NVIDIA. 2023. TensorRT-LLM. https://github.com/NVIDIA/TensorRT-LLM.

[37] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139* (2017).

[38] 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 (2023).

[39] Markus N Rabe and Charles Staats. 2021. Self-attention Does Not Need $O(n^2)$ Memory. *arXiv preprint arXiv:2112.05682* (2021).

[40] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. ZeRO-Offload: Democratizing Billion-Scale model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 551–564.

[41] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. 2021. INFaaS: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 397–411.

[42] ShareGPT. 2023. https://sharegpt.com/.

[43] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: A GPU cluster engine for accelerating DNN-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 322–337.

[44] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E Gonzalez, et al. 2023. High-throughput generative inference of large language models with a single gpu. *arXiv preprint arXiv:2303.06865* (2023).

[45] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[46] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[47] 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* (2023).

[48] 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 (2017).

[49] Xiaohui Wang, Ying Xiong, Yang Wei, Mingxuan Wang, and Lei Li. 2021. LightSeq: A High Performance Inference Library for Transformers. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Industry Papers (NAACL-HLT)*. Association for Computational Linguistics, 113–120.

[50] Yiding Wang, Kai Chen, Haisheng Tan, and Kun Guo. 2023. Tabi: An Efficient Multi-Level Inference System for Large Language Models. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 233–248.

[51] 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. *International Conference on Machine Learning (ICML)* (2023).

[52] 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)*. 521–538.

[53] Lingfan Yu and Jinyang Li. 2023. Stateful Large Language Model Serving with Pensieve. In *arXiv preprint 2312.05516*.

[54] Biao Zhang and Rico Sennrich. 2019. Root mean square layer normalization. *Advances in Neural Information Processing Systems* 32 (2019).

[55] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. 2022. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068* (2022).

[56] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2023. SGLang: Efficient Execution of Structured Language Model Programs.

[57] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *OSDI*.