Infinite-LLM: Efficient LLM Service for Long Context with DistAttention and Distributed KVCache

Bin Lin* Chen Zhang* [†] Alibaba Group Shanghai Jiao Tong University		Tao Peng* Alibaba Group	
Hanyu Zhao	Wencong Xiao	Minmin Sun	
Alibaba Group	Alibaba Group	Alibaba Group	
Anmin Liu	Zhipeng Zhang	Lanbo Li	
Peking University	Alibaba Group	Alibaba Group	
Xiafei Qiu	Shen Li	Zhigang Ji	
Alibaba Group	Alibaba Group	Shanghai Jiao Tong University	
Tao Xie	Yong Li	Wei Lin	
Peking University	Alibaba Group	Alibaba Group	

Abstract

Large Language Models (LLMs) demonstrate substantial potential across a diverse array of domains via request serving. However, as trends continue to push for expanding context sizes, the autoregressive nature of LLMs results in highly dynamic behavior of the attention layers, showcasing significant differences in computational characteristics and memory requirements from the non-attention layers. This presents substantial challenges for resource management and performance optimization in service systems. Existing static model parallelism and resource allocation strategies fall short when dealing with this dynamicity. To address the issue, we propose Infinite-LLM, a novel LLM serving system designed to effectively handle dynamic context lengths. Infinite-LLM disaggregates attention layers from an LLM's inference process, facilitating flexible and independent resource scheduling that optimizes computational performance and enhances memory utilization jointly. By leveraging a pooled GPU memory strategy across a cluster, Infinite-LLM not only significantly boosts system throughput but also supports extensive context lengths. Evaluated on a dataset with context lengths ranging from a few to 2000K tokens across a cluster with 32 A100 GPUs, Infinite-LLM demonstrates throughput improvement of 1.35-3.4x compared to state-of-the-art methods, enabling efficient and elastic LLM deployment.

1 Introduction

Large Language Models (LLMs)[5, 9, 15, 45, 53] have significantly advanced the field of generative artificial intelligence, and these inspiring capabilities have been integrated into various aspects of daily life. The universality of LLMs is

evident across numerous domains, such as programming copilot[16, 22], document summarization[50, 57], information retrieval[46, 60], and chatbots[25, 52]. The inference serving of LLMs [2, 6, 29] has emerged as a critical component within cloud infrastructures.

Today's LLM serving typically employs autoregressive mechanisms[14, 38, 44, 49] to iteratively generate output tokens and intermediate contexts (a.k.a. KVCache[36]). The autoregressive nature of these models introduces a characteristic of unpredictability in the sequence of generated tokens, as the process continues until the generation of an end token. As a result, the required memory and computational resources for LLM services dynamically change, with neither the lifetime nor the length of the context known a priori. With the rapid development of LLMs, the supported context is continuously expanding[10]. Multiple LLM vendors have significantly increased their capacity to millions of tokens—e.g., 128K for ChatGPT [35], 1000K for Google's Gemini [21], and 2000K for LongRoPE [19].

In LLM cloud service systems, resource demands are highly dynamic due to the enormous dynamicity and unpredictability of context generation tasks for LLMs. Since context generation tasks may generate arbitrary lengths, from as few as 1 to up to 2000K tokens, the cloud services must cater to a broad range of demands. Due to the unpredictable length of context generated by each request, pre-assigning resources accurately becomes unfeasible, leading to highly varied demands for computing and memory resources. For example, a single instance (i.e., a model replica deployed to handle request data in parallel) might manage numerous compute-intensive short-context tasks at one time and switch to memory-intensive long-context tasks or a mix of varying lengths at another

The preceding complexity in dynamic resource demands results in LLM service systems that struggle to *efficiently* and

^{*}Equal contribution.

[†]Corresponding author: chenzhang.sjtu@sjtu.edu.cn.

elastically adapt to varying workload requirements under different context lengths. This often leads to reduced system efficiency, manifesting primarily in two aspects:

Inefficient Model Parallelism inside an Instance. The model parallelism strategy required for processing requests with short and normal-length contexts differs significantly from that for long contexts. Traditional LLM service systems use a fixed model parallelism, where each instance is allocated a fixed number of GPUs. This fixed allocation makes it challenging to flexibly support both long and normal-length contexts efficiently. For example, processing a context with a length of 1K tokens on the Llama-7B model requires approximately 15 GB of memory, a fraction of an A100 GPU's capacity, while a context with a length of 1000K tokens demands over 500 GB, equivalent to the combined memory of about 7 A100 GPUs. Consequently, a higher degree of parallelism (DoP) is necessary for longer requests to meet their extensive resource needs, in stark contrast to the minimal requirements of shorter tasks. Configuring the system to meet the high DoP needed for long requests results in excessive model slicing and communication overhead for shorter requests, severely impacting performance. Existing parallelism strategies such as Tensor Parallelism and Pipeline Parallelism [34, 42], which are based on static model dimensions, struggle to adapt flexibly to such dynamic workloads.

Inefficient Resource Management across Instances. The dynamic lengths of requests also limit the efficiency of resource management across instances and the cluster throughput. In particular, it is difficult for the scheduler to find an optimal request placement to saturate both memory and compute utilization. This is because the memory utilization is determined only by the total KVCache, whereas the compute utilization largely depends on the batch size, i.e., number of running requests. For example, when a request grows too long on an instance, its KVCache will consume too much memory space on that instance, which in turn will greatly reduce the running batch size and compute utilization, even the memory is saturated. Similarly, when requests are short on an instance, the spare memory also cannot be harvested by the long requests on other instances. As a result, the overall cluster throughput would be limited.

Through an in-depth analysis of the computational characteristics of LLM models, we identified that the root of the challenges lies in the significant differences between attention and non-attention layers: non-attention layers exhibit static behavior with changing sequence lengths and are sensitive to batch size, while attention layers display dynamic behavior and are not affected by the batch size. To address these challenges, we present Infinite-LLM, a novel LLM service system designed for managing highly dynamic context lengths in LLM requests. Infinite-LLM introduces a new approach that decouples the computation of attention layers from the rest of the LLM model. This decoupling allows for flexible and independent resource scheduling, specifically

targeting the memory needs of dynamic attention layers and computation needs of the rest of the LLM model. Additionally, Infinite-LLM optimizes resource allocation by using the entire cluster's GPU memory as a pooled resource, allowing instances with surplus memory capacity to aid those processing extensive context tasks. This method not only significantly enhances resource efficiency and system throughput but also enables the cluster to support tasks with extremely long context lengths that surpass the memory limits of a single instance.

The contributions of this paper are summarized as follows:

- We reveal the dynamic characteristic of LLM request serving, and identify the limitations inherent in existing static model parallelism deployments and KV-Cache scheduling within a single instance.
- We present DistAttention, a novel attention mechanism that is mathematically equivalent to the original attention, designed to flexibly disaggregate attention computation and KVCache in a distributed way.
- We propose Infinite-LLM, an efficient LLM serving system specifically designed to adapt to the LLM serving dynamicity. It is capable of supporting scalable context length efficiently, by scheduling KVCache in cluster-level, thus to balance resource requirements between instances and achieve high overall system throughput.
- Evaluations show that Infinite-LLM can serve 2,000K tokens with 32 GPUs, achieving 1.35-3.4× improvement in end-to-end performance compared to stateof-the-art LLM serving system.

2 Background and Motivation

2.1 LLM Serving and Parallelism Method

LLM Inference. Large Language Models (LLMs)[9, 15, 45] are dominated by Transformer architectures [14, 49]. A Transformer block consists of three key components: the QKV Linear layer, Multi-Head Attention Mechanism, and Feed-Forward Neural Network (FFN) modules. Multi-Head Attention involves attention kernels and persists key-value cache (KVCache) across iterations, while both the OKV Linear layers and FFN layers are mostly General Matrix to Matrix Multiplication (GEMM) kernels. Inference serving of LLMs is autoregressive. Specially, the prefill phase takes prompts as inputs to generate the first token, and each new token afterwards is generative iteratively until an "end-of-sequence" (EOS) token, usually referred to as decode. Due to the generality of LLM, context length of inference serving can be wide-ranging [3, 7, 11, 56], from only 1 token [51] to 2000K tokens [19]. To further scale the serving throughput capacity, multiple model replicas are deployed to handle request data in parallel (a.k.a. data-parallel). Deployed as an instance, each replica contains a full copy of model parameters. This

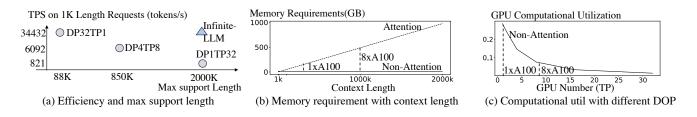


Figure 1. Static model parallelism struggles to maintain efficiency across all context length

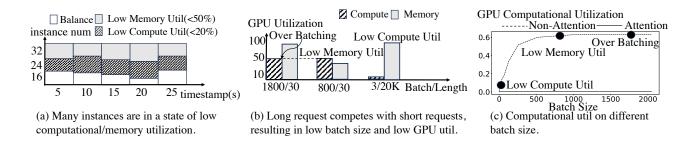


Figure 2. Resource under-utilization across instances in a cluster

instance is responsible for accommodating an LLM request and processing it to completion iteratively.

Model Parallelism. Model-parallel is necessary for supporting large models that do not fit into the memory of a single device. It can expand the total memory available to the serving instance for storing its prompt inputs, model weights, and intermediate values. Tensor parallelism and pipeline parallelism are the two major categories. Tensor-parallel [42] partitions a model layer across multiple GPUs, with each GPU executing a part of inference computation in parallel. Communication is required to split input or merge output for subsequent layers in the model. Pipeline-parallel [29, 33] avoids intra-layer communication by assigning contiguous layers on different GPUs for pipeline fashion execution, introducing inter-layer communication instead.

2.2 System Design Challenges

In this section, we delve into the computational characteristics of attention and non-attention layers at various text lengths and the challenges that these differences pose to system design. To better illustrate our points, we conduct a series of motivating experiments using the LLaMA2-7B model on 32 A100 GPUs.

Observation 1: Instances with a higher number of GPUs are capable of supporting long-text tasks but perform poorly on normal-text tasks. As illustrated in Figure 1(a), the performance differences across these instances are stark: Instance DP1TP32, with the most GPUs in a single instance (32 GPUs), has the largest memory capacity to support text generation tasks up to 2000K in length but performs the

worst on text generation tasks of standard length (1K). Conversely, as the number of GPUs decreases in Instances DP4TP8 (with 8 A100 each) and DP32TP1 (with only 1 A100 each), the maximum supportable text length decreases, while performance on text generation tasks of standard length improves.

This phenomenon originates from the different computational characteristics of attention and non-attention lavers across context lengths of high dynamic range. As shown in Figure 1(b), the tensor size of attention layers grows steadily with the context length, thus requiring more memory space and a higher degree of parallelism, typically necessitating deployment across more GPUs. For example, supporting a single text generation task of 1000K length would require at least 8 A100 GPUs. In contrast, the tensor size of nonattention layers does not change with text length; hence, no GPU number increase is needed. Traditional LLM parallel strategies [23, 42] do not differentiate between attention and non-attention layers, applying static model splits such as tensor or pipeline parallelism indiscriminately. This nondifferentiation can lead to non-attention layers being mapped to an excessive number of GPUs, potentially reducing computational performance due to over-segmentation. As shown in Figure 1(c), for a text generation task of 1000K length, the performance of non-attention layers on an 8 GPU instances is only about one-third of that on a single GPU instance.

Observation 2: When handling tasks with long context lengths, the computational utilization of GPU significantly decreases, and for short contexts, there is insufficient GPU memory utilization. This leads to instances typically exhibiting low computational or low memory utilization during the service process, as illustrated in

Figure 2(a). To illustrate this, we analyzed the performance of the decode phase for a single A100 LLaMA2-7B instance in a simplified scenario where all requests have the same context length. The results, depicted in Figure 2(b), show that when the context length is 20, the memory can support up to 1800 requests. However, this situation indicates over batching; compared to a batch size of 800, there is hardly any improvement in GPU compute utilization, suggesting that the additional 1000 requests do not contribute to performance improvement but instead significantly increase the request's latency. In the appropriate case with a batch size of 800, the actual GPU memory utilization is only 42%. Given that single A100 GPUs have a fixed total memory capacity of 80GB, as the context length of requests increases, the system is forced to handle smaller batches. Specifically, when the context length reaches 20K, the maximum batch size is limited to 3, leading to GPU compute utilization being only one percent of what it is when the context length is 20.

This issue is rooted in the stark contrasts in computational characteristics and resource demands between attention and non-attention layers. Non-attention layers utilize weight parameters that can be shared across all input vectors for requests. This capability allows the system to transform 'matrix-vector' multiplications (GEMV) into more efficient 'matrix-matrix' multiplications (GEMM) as batch sizes increase, significantly improving the computation-to-memory ratio and achieving high computational utilization, as shown in Figure 2(c). With the continued growth in context length, LLM service systems are forced to reduce batch sizes to free up memory to accommodate the increasing KVCache of the attention layers, resulting in decreased GPU compute utilization. However, in previous systems [4, 26, 41, 55], requests could only utilize the fixed resources available within their instance, limiting the system's ability to adapt to highly dynamic resource demands.

In summary, these observations highlight the need for adaptive parallelism and resource management strategies to efficiently handle the varying demands of different context lengths, optimizing both computational and memory resource utilization across large-scale clusters.

3 System Overview

The key concept of Infinite-LLM is to distribute the attention computation and KVCache beyond the boundaries of LLM inference instances, in order to leverage the resources of the entire GPU cluster, as illustrated in Figure 3. This idea disaggregates the attention layers from the non-attention layers, allowing them to employ independent parallel strategies and resource scheduling policies. It further enhances the scheduling strategy's ability to effectively manage the computation and memory of GPU resources at the cluster level.

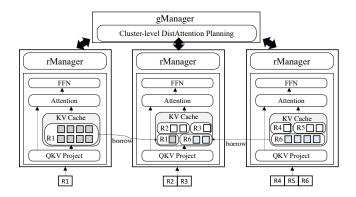


Figure 3. Infinite-LLM System Overview

Infinite-LLM accomplishes the design goal through three main system innovations. First, we introduce DistAttention, a novel attention mechanism that subdivides attention computations across GPUs, meanwhile avoiding KV-Cache transfer at decoding. DistAttention is mathematically equivalent to the common attention modules, such as multihead attention, multi-query attention, and grouped-query attention[39, 49]. The new distributed attention mechanism allows partition the attention in arbitrary size of sequence length and introduces negligible output data to be transferred for the other layers. Therefore, attention can be efficiently disaggregated in a scalable way (Section 4). Second, DistAttention can be utilized in multiple ways to resolve the resource contention and low efficiency of processing long requests and small requests thereby to significantly improve the cluster throughput. We model the major attention and non-attention cost of using DistAttention to formulate the aggregated cluster throughput. Achieving the optimal cluster throughput is costly as LLM serving is dynamic and unpredictable. Infinite-LLM includes a greedy scheduling policy based on our empirical study, approaching the improved cluster throughput and efficiency (Section 5). Third, Infinite-LLM introduces a new centralized controller, gManager, to host the scheduling policy and coordinate the dynamic interinstance KVCache tracking and migration. For scalability and fault tolerance, gManager works with a series of rManagers in a distributed architecture through a set of newly defined protocol (Section 6).

4 DistAttention

To achieve dynamic and flexible management of the KV-Cache, we propose DistAttention, a method that subdivides attention and the KVCache into regular small sub-blocks, thereby allowing the attention layer to be efficiently distributed and computed across multiple instances. Unlike traditional model parallelism methods, DistAttention is characterized by its slicing along the dynamic sequence dimension of the KVCache, allowing newly generated tokens in the auto-regressive process to be flexibly grouped, scheduled,

4

and computed. Although the KVCache tensor in the original attention can also be partitioned along the sequence dimension, the complex computation pattern of attention means that direct partition introduces significant communication overhead, greatly affecting the computational efficiency of distributed Attention. Inspired by online softmax[37], DistAttention successfully addresses this issue through an equivalent mathematical transformation on the original attention. Equation 1 shows the original computation formula of attention, requiring calculating the maximum attention score $(m_q \text{ in Equation 1})$ across all sequences and summing the intermediate results along the sequence dimension, thus necessitating the entire KVCache of all sequences. If KVCache is directly partitioned and stored in a distributed manner, it would necessitate transferring the KVCache from remote instances back to the local machine for each attention computation. As illustrated in Figure 4(a), given the substantial size of the KVCache, each decoding step needs to transfer GBs or even TBs of data, significantly impacting the performance of distributed attention computation.

$$m_{g} = max(QK_{1}, ..., QK_{seq})$$
Attention(Q, K, V) =
$$\sum_{i=1}^{seq} \frac{\exp(QK_{i}^{T} - m_{g})}{\sum_{j=1}^{seq} \exp(QK_{j}^{T} - m_{g})} V_{i}$$
 (1)

DistAttention's equivalent mathematical transformation on the original attention avoids the need to perform max and summation operations across all sequences. It allows each instance to execute the max and summation operations locally on partial KVCache data with partial sequence length seq_p . As shown in Equation 2, MicroAttention (MA) refers to the partial attention computations that result from the partition and can be distributed across various instances for computation. Consequently, for each distributed computation of attention, instances need to transfer the query vector along with only two float values, e_j and m_j .

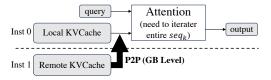
$$m_{j} = max(QK_{1}, ..., QK_{seq_{p}}), e_{j} = \sum_{i=1}^{seq_{p}} \exp(QK_{i}^{T} - m_{j})$$

$$MA_{j}(Q, K, V) = \sum_{i=1}^{seq_{p}} (\exp(QK_{i}^{T} - m_{j})V_{i})$$

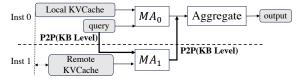
$$m_{g} = max(m_{1}, ..., m_{b}), e_{g} = \sum_{j=1}^{b} e_{j} \exp(m_{j} - m_{g})$$

$$Attention(Q, K, V) = \sum_{i=1}^{b} \frac{MA_{j} \exp(m_{j} - m_{g})}{e_{q}}$$
(3)

The intermediate results computed by the b remote instances are then transferred back to the local instance for aggregation (Equation 3) to arrive at an outcome equivalent to the original attention, as indicated in Figure 4(b). Because the computation FLOPs for aggregation is less than 1% of the total MA computational load, this overhead is virtually negligible. Since the query round trip involves only a few KBs



(a) Original attention necessitates traversing the full sequence, thus requiring communication of KVCache.



(b) DistAttention achieves communication of only the query through equivalent mathematical transformations on original attention.

Context len	8192	16384	32768	65536	131072
ship query	0.075ms	0.081ms	0.12ms	0.21ms	0.36ms
ship kvcache	0.581ms	1.080ms	1.98ms	3.77ms	7.48ms

(c) Time comparison between shipping KVCache and shipping the query.

Figure 4. DistAttention reduces the communication overhead through equivalent mathematical transformations.

of data, DistAttention substantially reduces the data communication overhead, as depicted in Figure 4(c).

5 Cluster-scale Throughput Optimization

5.1 Overview

DistAttention allows Infinite-LLM to place and compute a single request across multiple instances. This is not merely a means to enable serving extremely long requests beyond a single instance's capacity; we show that, from a cluster perspective, DistAttention is a powerful weapon as it greatly enlarges the request scheduling space across instances and can improve cluster-wide throughput. In particular, with DistAttention, Infinite-LLM is no longer limited to scheduling KVCache of each whole request on an instance; instead, Infinite-LLM can schedule any arbitrary sub-blocks of request's KVCache onto instances, which represents a much finer scheduling granularity and higher flexibility than existing systems.

Such sub-block level scheduling is beneficial because Infinite-LLM can better balance the KVCache and the batch sizes across instances, thereby maximizing memory and compute utilization simultaneously. Specifically, Infinite-LLM maintains balanced batch sizes by controlling the number of sub-blocks on each instance, preventing individual requests from occupying too much memory and decreasing the batch size. Figure 5 shows an intuitive example. Figure 5(a) demonstrates the initial status on the KVCache distribution of four serving instances. Instance A is processing a long request that saturates all memory space, and Instance D is processing a long request however with some available GPU memory

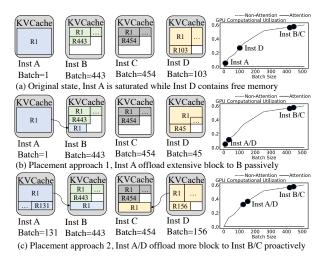


Figure 5. Effects of two different placement methods

left. Instances B and C are handling short requests, and despite high batch sizes, they still have ample memory space. Figure 5(b) shows a strawman placement strategy that only places the newly generated blocks onto the instance with the most remaining space when the length of a long request exceeds the memory capacity of an instance. While this approach can support long requests that exceed the resource capacity of a single instance, the cluster as a whole maintains a relatively low throughput: the batch size for Instance A remains at 1, and the newly generated attention sub-blocks of the long request in Instance D compete with local short requests, reducing the batch size and resulting in low GPU compute utilization. The second placement method (Figure 5(c)) proactively places more sub-blocks onto other instances with sufficient available space before the length of a long request exceeds the instance memory capacity. As shown in Figure 5(c), Instances A and D proactively place more attention sub-blocks to Instances B and C, freeing up memory space to handle more short requests, thereby increasing the batch size. Compared to the first method, this proactive placement balances the batch sizes among instances, improving the overall cluster throughput.

The insights summarized from the above study inspire us to devise a reasonable scheduling method for placing attention sub-blocks to enhance the overall throughput of the cluster as much as possible. Our scheduling method must address three main issues: (1) If an instance offloads part of its KVCache to remote, how to determine a proper size? What is the performance gain and overhead? (2) If an instance lends some space out, how much space should be used? (3) Given that there are numerous instances, how to decide the borrowlend relationship that maximizes the overall performance?

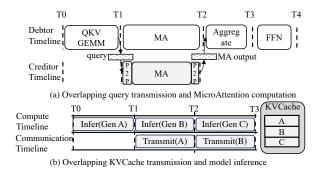


Figure 6. Communication Overlapping Optimization

5.2 Debtors and Creditors

To address these problems, we discuss the performance impacts between instances that borrow or lend memory spaces. We refer to instances that borrow memory from others as *debtors*, such as Instances A and D, and instances that lend memory as *creditors*, such as Instances B and C in Figure 5. We do not allow an instance to act both as a debtor and a creditor simultaneously. Whenever a debtor has free local space (e.g., a request retires and frees up its memory), it prefers to increase its batch size, or to retrieve attention subblocks that were previously offloaded instead of lending it to others. Similarly, a creditor, lacking sufficient local space, will reclaim the memory that it has previously lent out.

5.2.1 Debtors Debtors borrow memory space from one or more creditors to store portions of their KVCache. This operation has both positive and negative impacts on their performance. The primary benefit is that offloading part of the attention computation to creditors reduces the time debtors spend generating tokens, and the freed-up space allows higher batch size, thereby improving generation throughput.

This approach also introduces several challenges. First, debtors must collect the partial attention computation results from creditors to complete the attention calculation. If creditors compute an excessive number of MicroAttention (MA) operations, it may lead to idle waiting for the debtors and thus reduce performance. As shown in Figure 6(a), to mitigate this issue, our scheduling policy aims to limit the size of KVCache on remote instances so that the remote computation and transmission can be entirely covered by local computations. Secondly, transferring the KVCache to creditors is time-consuming. To minimize the impact of KVCache transfer on LLM inference performance, as shown in Figure 6(b), we overlap the transfer with local model inference. Micro-benchmarking and performance analysis. Figure 7(a) depicts the debtor's throughput as a function of the number of KVCache blocks moved to the creditor, represented by the dashed line. As more KVCache are offloaded, the debtor's throughput greatly increases because the debtor enjoys a boost in performance when its batch size is very

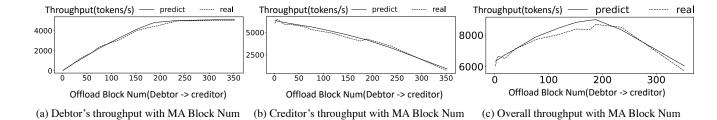


Figure 7. Experiment is conducted using LLAMA2-7B model on two A100 instances. One instance acts as a creditor executing normal-length requests (500 tokens on averag). The other acts as a debtor, processing a long context (1000K tokens).

low. As the batched requests approach the computational limits of the system, this trend eventually plateaus.

5.2.2 Creditors Creditors lend their excess memory space to one or multiple debtors. Due to the additional computation of partial attention for debtors, the performance of creditors' local request services is negatively affected.

Micro-benchmarking and performance analysis. Figure 7(b) reveals a slow but steady performance degradation with more KVCache moved to the creditor. When the transferred KVCache exceeds the surplus memory space of the creditor, the batch size of the creditor is reduced, resulting in an even steeper decline in performance.

5.2.3 Overall cluster throughput The overall system throughput is obtained by summing the throughput of all instances in the cluster. Our optimization goal is to find an effective KVCache placement strategy that enhances the throughput of the entire system. For example, Figure 7(c) presents the aggregated performance of a micro-benchmarking test for a debtor and a creditor. As the KVCache is transferred, the total throughput increases from approximately 6500 tokens/second to about 8800 tokens/second. As more MA blocks are moved to the creditor, the overall throughput sharply declines. Under this configuration, the system achieves maximum aggregate throughput when 200 blocks of KVCache are transferred.

Complexity analysis. However, determining a KVCache placement schedule for the real cluster is very difficult because the design space is prohibitively complex. Considering a cluster with N instances as debtors and M ones as creditors, we refer the number of surplus memory blocks in each creditor as Y_i . For each block, there are N+1 possible options: offering it to one of the N debtors or to not lend it out at all. Supposing all memory blocks make the decision independently, the search space can be $(N+1)^{\sum_{i=1}^{M} Y_i}$. However, the blocks within each creditor are homogeneous. After deduplication, the final search space size is as follows:

$$\frac{(N+1)^{\sum_{i=1}^{M} Y_i}}{\prod_{i=1}^{M} Y_i!} \tag{4}$$

Such a huge design space makes it impractical to figure out the optimal cluster throughput during runtime. Next, to avoid the overhead of empirical measurements, we have constructed a performance model to predict the overall cluster throughput for a given placement strategy, and we propose an optimization algorithm based on this model.

5.3 Scheduling Algorithm

To efficiently figure out an efficient KVCache placement schedule during runtime, we propose a greedy algorithm based on a performance model.

Performance modeling. Equation 5 outlines a general analytical model for single transformer layer, comprising both non-attention and attention layers. The computational load of all non-attention layers, denoted as $W(\beta)$, is primarily influenced by batch size β . The GPU's real performance(FLOPs/s), denoted as $f(\beta)$, is closely tied to batch size and can be experimentally measured. The workload of attention layers is dictated by the requests' length S. Since attention layers cannot benefit from batching, their GPU performance, denoted as g(S), typically remains constant and is also ascertainable through experimental methods.

$$T^{lyr}(\beta, S) = T^{natn}(\beta) + T^{atn}(S) = \frac{W(\beta)}{f(\beta)} + \sum_{r=1}^{\beta} \frac{S^r}{g(S)}$$
 (5)

Equation 6 extends this performance model to specifically address the roles of debtors and creditors within the system. In Infinite-LLM, a debtor can offload KVCaches of size K^d to creditors, allowing an increase in its batch size to β' . Meanwhile, a creditor may allocate space for KVCaches of size K^c to debtors while maintaining its original batch size β .

$$T_{dbt}^{lyr}(\beta',K^d)=T^{lyr}-\frac{K^d}{g(S)},T_{cdt}^{lyr}(\beta,K^c)= T^{lyr}+\frac{K^c}{g(S)} \eqno(6)$$

Combining the formulations above, per-instance throughput (a.k.a, tokens per second) is as $TPS = \frac{\beta}{n \cdot T^{layer}}$, where n represents the number of transformer layers of an LLM. For a cluster deployed with M instances, the overall aggregated

7

throughput equals to the sum of the *TPS* of all instances:

$$TPS_{cluster} = \sum_{i=1}^{M} TPS_i \tag{7}$$

We have validated the accuracy of the performance model, which is shown in Figure 7. The results predicted by the performance model are consistent with the real measurements. Having obtained the formula for calculating the overall cluster throughput, we can estimate the cluster throughput with any specific schedules.

Greedy Algorithm. We propose a greedy algorithm to maximize the cluster throughput with DistAttention approximately. Our algorithm is founded on a principle: pairing overloaded debtor instances with the free creditor to continuously perform load balancing scheduling, thereby enhancing overall throughput.

As illustrated in Algorithm 1, we select instances with batch sizes smaller than the empirical threshold β^{thres} as debtor instances, while those with memory utilization rates below the empirical threshold U^{thres} serve as creditor instances. As small batch size instances are with great performance potential empirically, debtors are processed in ascending order according to their batch sizes. At each round, the longest request r is selected from the debtor, as well as the creditor with maximal available memory. The possible block number to move for request r is explored to estimate the potential throughput gain. Specifically, we first establish the upper limit on the number of MA blocks that can be offloaded, which corresponds to the number of MA blocks for request r. Under the constraint of maximum number of MA Blocks, $Block_{max}$, the performance model of aggregated throughput of two instances (as illustrated in Equation 7) is utilized to determine the number of offload blocks between 0 and $Block_{max}$. For each debtor, the algorithm loops the creditors in orders until no performance gain can be achieved from the memory block movement. In Infinite-LLM, the algorithm acts periodically and retrospecively to adapt to the dynamic and unpredictable serving load of LLM.

6 System Design

6.1 gManager and rManager

To realize the global planning described in section 5, Infinite-LLM employs a centralized manager termed the *gManager* to maintain a global view of instance status and make the request and KVCache placement decisions. The gManager tracks the KVCache placement on the instances of each request, maintained in the *request placement map*, where each entry represents (part of) the KVCache memory usage of a request on a certain instance. A request is allowed to be distributed on multiple instances, i.e., having multiple entries. One of the entries/instances of a request is marked as the debtor instance of it. Utilizing this data, gManager tracks the

Algorithm 1 Cluster-level DistAttention Scheduling

```
1: Collect debtors with small batch size D^i \in \{I^i, \beta^i \leq \beta^{thres}\}
 2: Sort debtors in increased batch size order \langle I^i, \beta^i \rangle
 3: Collect creditors with low memory util C^i \in \{I^i, U^i \leq U^{thres}\}
 4: Sort creditors in increased memory util order \langle I^i, U^i \rangle
 5: for D^i \in D do
         r = \text{pick\_longest\_request}(D^i)
 6:
         Block_{max} = get\_block\_num(r)
 7:
         for C^j \in C do
 8:
              for k \in \text{range}(0, Block_{max}) do
 9:
                   \langle perf, k \rangle = perf model throughput esti(k, D^i, C^j)
10:
11:
              Block_{best} = \mathrm{pick\_max\_perf}(\langle perf, k \rangle)
12:
              if Block_{best} \le 0 then
13:
                   break;
14:
15:
              move_kvcache(D^i, C^j, Block_{best})
16:
              update_and_sort_mem_util(C)
17:
              Block_{max} - = Block_{best}
18:
19:
         end for
20: end for
```

current status of request placements and then derives a new expected placement status and the transition plan.

Considering the rapid changing memory usages of requests, tracking the status of every single request precisely in the gManager would be prohibitively expensive. Infinite-LLM develops a distributed and coordinated system architecture to implement the global planning efficiently. As shown in Figure 3, Infinite-LLM introduces a series of distributed rManagers co-located with the instances. The gManager and rManagers work in a loosely-coordinated manner. That is, instead of keeping the global view in sync with the real request statuses, it relies on periodic heartbeat signals from the rManager of each instance to convey updates about the KVCache memory usages of requests on it. This approach reduces the overhead of the global planning and enhances system performance. After receiving a full update from all instances in each round, the gManager calculates a status transition plan and finally instructs the rManagers to move the KVCaches if needed.

Under this architecture, Infinite-LLM needs to deal with the potential staleness of the global view during the periodic updates. In normal cases, the running batch on each instance and its KVCache memory usage keep growing as the decoding computation proceeds after a periodic status update. Moreover, due to the continuous batching behavior, the memory usage may experience a steeper change when certain requests complete or new requests join. We design a protocol among the gManager and rManagers to implement this interaction.

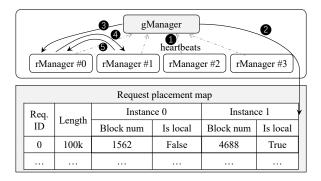


Figure 8. Overall workflow of Infinite-LLM's protocol

6.2 Protocol

Figure 8 shows the overall workflow of Infinite-LLM's protocol. We summarize the APIs used in the protocol in Listing 1. Each rManager reports its local status using the heartbeat API 1, which includes an array of the request placement entries (shown by the RequestPlacementEntry struct). Note that when a request locates in multiple instances, it is possible that its status on certain instances do not change in a period if the newly generated KVCaches are not placed on them. Therefore, in normal cases, the rManager only sends the entries that have changed since the last update to the gManager. An exception is that when initializing a new gManager (e.g., after a failover), the rManager will send the full information to help the gManager construct the initial status. The gManager updates these entries into the global request placement map accordingly **2**. The gManager then dispatches its request placement decisions using the move_kvcache API, which instructs an instance to move a certain amount of KVCache blocks to a destination instance 3

```
class RequestPlacementEntry:
    req_id:int, inst_id:int, num_blocks:int, local
    :bool
heartbeat(List[RequestPlacementEntry]) -> None
move_kvcache(req_id:int, num_blocks:int, dst_inst:
    int) -> None
try_move_kvcache(req_id:int, num_blocks:int)->bool
```

Listing 1. Infinite-LLM APIs

Considering the potential staleness of the request placement map, the instruction from the gManager to move KV-Cache to another instance could be infeasible — for example, when the KVCache on the destination instance grows and the memory space becomes insufficient. Therefore, Infinite-LLM further provides the try_move_kvcache API for the source instance to try to reserve space on the destination instance before transferring the real KVCache data ④. On the destination side, it may receive multiple concurrent try_move_kvcache calls from other instances. The destination instance uses a first-come-first-serve policy to decide the allocation among these competing candidates; if the total space is not enough

Trace		Range	Avg.	SD
	0	1-60k	1233	7785.68
S	1	1-60k	712	5531.4
	2	1-60k	469	3506.36
	3	1-200k	56362	28787.78
	4	1-280k	75650	39479.42
I.	5	1-600k	160239	87906.67
L	6	1-480k	128804	70647.93
	7	1-1200k	293945	172169.14
	8	1-2000k	498609	261817.24

Table 1. Ranges, average values, and standard deviations (SDs) of context lengths of the traces.

for satisfying all of the them, the destination instance will reject some of them. The desination instance responds with a boolean value to identify whether this KVCache movement is allowed or rejected **5**. If it is allowed, the source instance proceeds to transfer the data; otherwise, it simply waits for further instructions from the gManager in future rounds, which will have captured the latest cluster status.

7 Evaluation

7.1 Experimental Setup

Environment. We deploy Infinite-LLM on a cluster with 4 nodes and 32 GPUs. Each node has 8xNVIDIA A100 (80GB) GPUs. The GPUs are connected via NVLink (600GB/s) within each node and via Ethernet (125MB/s) across nodes.

Models. Since most LLM models have similar backbone Transformer block, we choose one representative model family, LLaMA2[48] for evaluation. The LLaMA2 family contains three different model sizes: 7B, 13B and 70B. They use two popular attention architectures; the 7B and 13B models utilize Multi-head Attention (MHA)[49], while the 70B model employs Grouped-Query Attention (GQA)[39].

Traces. We generate 9 traces with different context length ranges and length distributions to comprehensively evaluate Infinite-LLM's end-to-end performance. Traces 0-2, marked as "S" (short) in Table 1, have relatively short sequence lengths that are guaranteed to fit in each instance when using vLLM (i.e., vLLM-multi). Requests of trace 0 come from the open-source dataset ShareGPT4[47], which contains conversations of GPT4 service. To assess the impact of different length distributions, particularly the variance of sequence lengths, we select a subset of data from ShareGPT4 to construct traces 1 and 2 with reduced standard deviations. Traces 3-8, marked as "L" (long) in Table 1, are used to evaluate Infinite-LLM using larger context length ranges, where requests of trace 3 come from open-source dataset L-Eval[8], and traces 4-8 are from the distribution of long requests from our online service. In each experiment, we assign an arrival time to each request using a Poisson distribution using varying request rates.

Comparison. Our evaluation focuses on comparing Infinite-LLM with static model parallelism and resource planning. To this end, we use vLLM[26], a state-of-the-art LLM serving engine using static model parallelism, as the primary baseline. Specifically, we compare the following approaches:

- Infinite-LLM: Given the total cluster resources, Infinite-LLM divides them into multiple model instances using an appropriate parallelism configuration (for non-attention computation) while scaling the attention computation across the instances. Infinite-LLM dispatches each request to the instance with the most free GPU memory.
- *vLLM-multi* (vLLM-M): vLLM with the same number and parallelism configuration of the instances as Infinite-LLM. It might fail to run some long requests due to limited per-instance memory capacity. vLLM-M uses the same dispatching policy as Infinite-LLM.
- vLLM-single (vLLM-S): a single instance containing all the cluster resources, so that vLLM can support the same sequence length ranges as Infinite-LLM. Note that vLLM only supports tensor parallelism, which is known to be less efficient than pipeline parallelism when distributed across machines[42]. We implement pipeline parallelism in vLLM for cross-machine communication when this instance needs to be distributed on multiple machines.

7.2 Context Length Performance

We first benchmark the performance of Infinite-LLM and the baselines when running requests with different context lengths. We use six context length ranges and three models. For each model and range, we test three specific context lengths: (1) a short context length (1k); (2) a length slightly exceeding the maximum length that vLLM-multi can support; and (3) the maximum length that Infinite-LLM supports given the cluster resources. For each data point, we measure the throughput of a largest batch of requests given the context length.

As shown in Figure 9, Infinite-LLM achieves the best of both long and short sequences. Compared to vLLM-multi, which is limited by per-instance memory, Infinite-LLM supports substantially longer context (2x-19x) while achieving comparable throughput on short sequence lengths. This improvement is attributed to Infinite-LLM's ability to efficiently coordinate memory and computation usage across all instances, while vLLM-multi is limited to the instance's private resource. Compared to vLLM-single, Infinite-LLM obtains 1.4x-5.3x higher throughput on short lengths while sustaining similar longest context lengths. This is because Infinite-LLM can maintain an efficient model parallelism strategy for FNN computations while vLLM-single has to partition the model into smaller segments across more GPUs,

which results in lower GPU computation utilization for the Non-Attention parts and more communication overhead.

7.3 End-to-end Serving Performance

Comparison with multiple small instances. We first compare Infinite-LLM with vLLM-multi, which launches multiple instances with the same parallelism configuration as Infinite-LLM. We conduct six experiments using Traces 0-2, where the sequence lengths won't exceed the limit of each single instance of vLLM. Figure 10a shows the throughputlatency variation when using different request rates. We compare the maximum achieved throughputs of Infinite-LLM and vLLM. The results demonstrate that Infinite-LLM gets a throughput improvement of approximately 1.35x-1.73x over vLLM. We further examine how the number of instances and traces' context length distribution affect performance gains. As depicted in the six sub-figures of Figure 10a, from left to right, the standard deviation of the traces' context length distribution decreases while the number of instances increases from top to bottom. We observe that the performance gains rise with the standard deviation (indicating a more uneven length distribution) and the number of instances . This is because a more uneven length distribution or a larger number of instances lead to greater variance in resource demands among different instances, enhancing the benefits of unified resource management across all instances.

Comparison with a single large instance. We use Traces 3-8 with longer context lengths to compare Infinite-LLM with vLLM-S, which allocates all GPUs to a single instance to accommodate sufficiently long sequences. The results shown in Figure 10b indicate that Infinite-LLM gets a 1.4x to 3.4x throughput gain over vLLM. From top to bottom and left to right in Figure 10b, we observe that Infinite-LLM's performance gains grow with the context length range expanding. This is attributed to vLLM's static model parallelism fragmenting the model across more GPUs, leading to reduced efficiency in the non-attention segments and significantly lowering the system's capability to process shorter request efficiently, whereas Infinite-LLM maintains an appropriate model parallelism strategy for the non-attention part, thereby preserving their performance.

7.4 Micro-benchmarks

Comparison with other long-context attention methods. We compared the performance of DistAttention, RingAttention, and TP (partition by numer of heads) within the context range from 4K to 256K, where the Attention computation is based on the dimensions of LLaMA2-13B with four GPUs. As shown in Figure 11, the results show that DistAttention is 1%-25% faster than TP due to its lower communication overhead. Compared to RingAttention, DistAttention is 7.7x-19.8x faster owing to the significantly higher communication overhead of RingAttention, which involves the transfer of

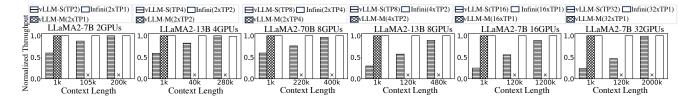
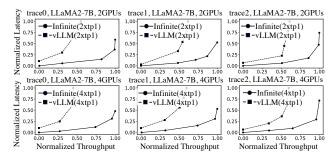
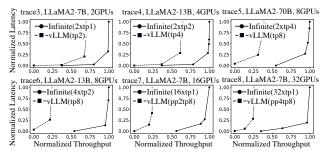


Figure 9. Context Length Performance



(a) Comparison with vLLM-M



(b) Comparison with vLLM-S

Figure 10. End-to-end serving performance.

large KVCache (MB to GB), whereas DistAttention transmits very small-sized queries (KB).

Overhead of KVCache movement. We improve cluster-scale throughput by scheduling DistAttention across all instances. To reduce the overhead of KVCache movement between instances, Infinite-LLM overlaps the movement with model computation. To evaluate the impact of movement communication on instance throughput, we compared the instance throughput with movement enabled to that with movement disabled. It's important to note that movement in this experiment does not change the batch size, hence any fluctuations in the throughput curve are due to the communication costs of movement. Results shown in Figure 12(a) indicate that instance throughput decreased by 8.6% when moving 32 tokens per decode step. When moving 16 tokens per decode step, the throughput of instance was identical to the instance with movement turned off. Therefore, when

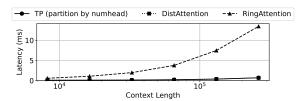


Figure 11. Comparison of distributed attention methods.

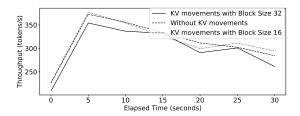


Figure 12. Overhead of KVCache movement.

the movement size is set to 16, communication can overlap well with computation without affecting the instance's performance.

8 Related Work

LLM inference system. ORCA [55] introduced iteration-level scheduling which greatly enhances the computation and memory utilization in batching inference. vLLM [26] further proposed PagedAttention to address the memory wastage due to fragmentation. DeepSpeed-FastGen[2] proposed a novel prompt and generation composition strategy called Dynamic SplitFuse (or Sarathi[4]) to further enhance system throughput. DistServe[59] proposed to disaggregate the prefill and decode stages to different instances to avoid their interference. Despite these novel systems solve many problems and achieve outstanding results, the dynamic problem along with the need to support exceptionally long context lengths still remains an unresolved challenge.

Long-context LLM. Works like FlashAttention[18] and FlashDecoding[1] focus on optimizing the performance of attention in long sequences. They enhance the compute-to-memory ratio and SM (Streaming Multiprocessors) parallelism of Attention on a single GPU by addressing data dependency issues. However, they do not take into account

the communication overhead in multi-GPU settings and cannot be directly applied to scenarios involving multiple GPUs. To train LLM with long context, some research work [13, 27, 28, 32] has introduced the method of context parallelism to partition the computation in sequence dimension. Ring Attention [31, 32] distributes long sequences across multiple devices, with the intent of fully overlapping the communication of KV blocks with the computation of blockwise attention. Those methods are designed for training, which is a poor fit to the highly dynamic characteristic in LLM inference decoding phase, causing substantial overhead to transfer KV-Cache across devices at each iteration. Another thread to address the challenge of oversized KVCache for long-context inference is to utilize sparse KVCaches such as Sliding Window Attention [12, 17, 24], H2O [58] and StreamingLLM [54], which compromises with the potential accuracy loss, because of the KVCache eviction. Infinite-LLM supports long-context LLM serving by introducing a new scalable distributed attention mechanism, DistAttention. Attention can be disaggregated from the model inference, therefore to schedule across multiple serving instances, both for computation and KVCache management. DistAttention retains equivalence to the original attention thereby be harmless to model accuracy. **Scheduling.** To improve throughput and latency of LLM serving, serveral systems [20, 30, 40, 43] have been proposed to optimize for request scheduling across multiple model instances. Llumnix[43] dynamically reschedules requests across multiple instances at runtime to deal with the heterogeneity and unpredictability of requests. Parrot[30] uncovers the dependencies and commonalities among LLM requests, thus creating a new space for enhancing the end-to-end performance of LLM applications. However, the previous work is limited to scheduling each whole request on an instance, facing issue of low GPU utilization due to dynamic request length. Infinite-LLM can schedule any arbitrary subsequences of requests onto instances, representing a much finer scheduling granularity and higher flexibility than existing systems.

9 Conclusion

In this paper, we have presented Infinite-LLM, a novel LLM service system designed for managing highly dynamic context lengths in LLM requests. Through Infinite-LLM, we have revealed the highly dynamic characteristic within LLM requests and advocated attention disaggregation to be a common technology for LLM serving. In particular, we have introduced a novel system architecture both efficient and scalable for all LLM requests, proposed a scheduling policy to saturate the computation and bandwidth of GPU simultaneously, and shown significant improvement through extensive evaluations on representative real traces. Going forward, we hope that Infinite-LLM can become a common foundation

toward AGI for both the research community and industry, inspiring future advancements in LLM serving.

References

- [1] Flashdecoding. https://princeton-nlp.github.io/flash-decoding/, 2021.
- [2] Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference. https://github.com/microsoft/DeepSpeed/ tree/master/blogs/deepspeed-fastgen, 2023.
- [3] Lisa Adams, Felix Busch, Tianyu Han, Jean-Baptiste Excoffier, Matthieu Ortala, Alexander Löser, Hugo JWL Aerts, Jakob Nikolas Kather, Daniel Truhn, and Keno Bressem. Longhealth: A question answering benchmark with long clinical documents. arXiv preprint arXiv:2401.14490, 2024.
- [4] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, and Ramachandran Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills, 2023.
- [5] AI@Meta. Llama 3 model card. 2024.
- [6] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–15. IEEE, 2022.
- [7] Chenxin An, Shansan Gong, Ming Zhong, Mukai Li, Jun Zhang, Lingpeng Kong, and Xipeng Qiu. L-eval: Instituting standardized evaluation for long context language models. arXiv preprint arXiv:2307.11088, 2023
- [8] Chenxin An, Shansan Gong, Ming Zhong, Xingjian Zhao, Mukai Li, Jun Zhang, Lingpeng Kong, and Xipeng Qiu. L-eval: Instituting standardized evaluation for long context language models, 2023.
- [9] Rohan Anil, Andrew M Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, et al. Palm 2 technical report. arXiv preprint arXiv:2305.10403, 2023.
- [10] Anthropic. https://www.anthropic.com/news/claude-3-family, 2024.
- [11] Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, et al. Longbench: A bilingual, multitask benchmark for long context understanding. arXiv preprint arXiv:2308.14508, 2023.
- [12] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. arXiv preprint arXiv:2004.05150, 2020.
- [13] William Brandon, Aniruddha Nrusimha, Kevin Qian, Zachary Ankner, Tian Jin, Zhiye Song, and Jonathan Ragan-Kelley. Striped attention: Faster ring attention for causal transformers. arXiv preprint arXiv:2311.09431, 2023.
- [14] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. Advances in neural information processing systems, 33:1877–1901, 2020.
- [15] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, Wei Ye, Yue Zhang, Yi Chang, Philip S. Yu, Qiang Yang, and Xing Xie. A survey on evaluation of large language models, 2023.
- [16] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin,

- Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.
- [17] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers, 2019.
- [18] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. Advances in Neural Information Processing Systems, 35:16344–16359, 2022.
- [19] Yiran Ding, Li Lyna Zhang, Chengruidong Zhang, Yuanyuan Xu, Ning Shang, Jiahang Xu, Fan Yang, and Mao Yang. Longrope: Extending llm context window beyond 2 million tokens. arXiv preprint arXiv:2402.13753, 2024.
- [20] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. Serverlessllm: Locality-enhanced serverless inference for large language models, 2024.
- [21] Google. Our next-generation model: Gemini 1.5. https://blog.google/technology/ai/google-gemini-next-generationmodel-february-2024/, 2024.
- [22] Dong Huang, Qingwen Bu, Jie M Zhang, Michael Luck, and Heming Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*, 2023.
- [23] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. Advances in neural information processing systems, 32, 2019.
- [24] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b, 2023.
- [25] Jin K Kim, Michael Chua, Mandy Rickard, and Armando Lorenzo. Chatgpt and large language model (llm) chatbots: The current state of acceptability and a proposal for guidelines on utilization in academic medicine. Journal of Pediatric Urology, 2023.
- [26] W Kwon, Z Li, S Zhuang, et al. Efficient memory management for large language model serving with pagedattention. In Proceedings of the 29th Symposium on Operating Systems Principles, pages 611–626, 2023
- [27] Dacheng Li, Rulin Shao, Anze Xie, Eric P Xing, Joseph E Gonzalez, Ion Stoica, Xuezhe Ma, and Hao Zhang. Lightseq: Sequence level parallelism for distributed training of long context transformers. arXiv preprint arXiv:2310.03294, 2023.
- [28] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. Sequence parallelism: Long sequence training from system perspective. arXiv preprint arXiv:2105.13120, 2021.
- [29] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. Alpaserve: Statistical multiplexing with model parallelism for deep learning serving. arXiv preprint arXiv:2302.11665, 2023.
- [30] Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. Parrot: Efficient serving of llm-based applications with semantic variable, 2024.
- [31] Hao Liu and Pieter Abbeel. Blockwise parallel transformer for long context large models. arXiv preprint arXiv:2305.19370, 2023.
- [32] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context. arXiv preprint arXiv:2310.01889, 2023.

- [33] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In Proceedings of the 27th ACM Symposium on Operating Systems Principles, pages 1–15, 2019.
- [34] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on GPU clusters. CoRR, abs/2104.04473, 2021.
- [35] OpenAI. https://openai.com/blog/chatgpt, 2022.
- [36] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. Proceedings of Machine Learning and Systems, 5, 2023.
- [37] Markus N Rabe and Charles Staats. Self-attention does not need $o(n^2)$ memory. arXiv preprint arXiv:2112.05682, 2021.
- [38] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.
- [39] Noam Shazeer. Fast transformer decoding: One write-head is all you need. arXiv preprint arXiv:1911.02150, 2019.
- [40] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E. Gonzalez, and Ion Stoica. Fairness in serving large language models, 2024.
- [41] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pages 31094–31116. PMLR, 2023.
- [42] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multibillion parameter language models using model parallelism, 2020.
- [43] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic scheduling for large language model serving, 2024.
- [44] Ilya Sutskever, James Martens, and Geoffrey E Hinton. Generating text with recurrent neural networks. In Proceedings of the 28th international conference on machine learning (ICML-11), pages 1017–1024, 2011.
- [45] Salmonn Talebi, Elizabeth Tong, and Mohammad RK Mofrad. Beyond the hype: Assessing the performance, trustworthiness, and clinical suitability of gpt3. 5. arXiv preprint arXiv:2306.15887, 2023.
- [46] Qiaoyu Tang, Jiawei Chen, Bowen Yu, Yaojie Lu, Cheng Fu, Haiyang Yu, Hongyu Lin, Fei Huang, Ben He, Xianpei Han, et al. Self-retrieval: Building an information retrieval system with one large language model. arXiv preprint arXiv:2403.00801, 2024.
- [47] ShareGPT Team. https://huggingface.co/datasets/shibing-624/sharegpt_gpt4, 2023.
- [48] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and finetuned chat models. arXiv preprint arXiv:2307.09288, 2023.
- [49] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. Advances in neural information processing systems, 30, 2017.
- [50] Yiming Wang, Zhuosheng Zhang, and Rui Wang. Element-aware summarization with large language models: Expert-aligned evaluation and chain-of-thought method. arXiv preprint arXiv:2305.13412, 2023.
- [51] Yuxin Wang, Yuhan Chen, Zeyu Li, Zhenheng Tang, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. Towards efficient and reliable llm serving: A real-world workload study, 2024.
- [52] Jing Wei, Sungdong Kim, Hyunhoon Jung, and Young-Ho Kim. Lever-aging large language models to power chatbots for collecting user self-reported data. arXiv preprint arXiv:2301.05843, 2023.

- [53] BigScience Workshop, Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, et al. Bloom: A 176bparameter open-access multilingual language model. arXiv preprint arXiv:2211.05100, 2022.
- [54] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. arXiv preprint arXiv:2309.17453, 2023.
- [55] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 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, 2022.
- [56] Tao Yuan, Xuefei Ning, Dong Zhou, Zhijie Yang, Shiyao Li, Minghui Zhuang, Zheyue Tan, Zhuyu Yao, Dahua Lin, Boxun Li, et al. Lv-eval: A balanced long-context benchmark with 5 length levels up to 256k. arXiv preprint arXiv:2402.05136, 2024.

- [57] Tianyi Zhang, Faisal Ladhak, Esin Durmus, Percy Liang, Kathleen McKeown, and Tatsunori B Hashimoto. Benchmarking large language models for news summarization. *Transactions of the Association for Computational Linguistics*, 12:39–57, 2024.
- [58] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen. H₂o: Heavy-hitter oracle for efficient generative inference of large language models, 2023.
- [59] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving, 2024.
- [60] Yutao Zhu, Huaying Yuan, Shuting Wang, Jiongnan Liu, Wenhan Liu, Chenlong Deng, Zhicheng Dou, and Ji-Rong Wen. Large language models for information retrieval: A survey. arXiv preprint arXiv:2308.07107, 2023