arXiv:2401.02669v1 [cs.DC] 5 Jan 2024

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

Bin Lin[1*], Tao Peng[1*], Chen Zhang[2*†], Minmin Sun[1], Lanbo Li[1], Hanyu Zhao[1], Wencong Xiao[1], Qi Xu[1], Xiafei Qiu[1], Shen Li [1], Zhigang Ji[2], Yong Li[1], Wei Lin[1]

[1]Alibaba Group, [2]Shanghai Jiao Tong University

Abstract

The rapid proliferation of Large Language Models (LLMs) has been a driving force in the growth of cloud-based LLM services, which are now integral to advancing AI applications. However, the dynamic auto-regressive nature of LLM service, along with the need to support exceptionally long context lengths, demands the flexible allocation and release of substantial resources. This presents considerable challenges in designing cloud-based LLM service systems, where inefficient management can lead to performance degradation or resource wastage. In response to these challenges, this work introduces `DistAttention`, a novel distributed attention algorithm that segments the KV Cache into smaller, manageable units, enabling distributed processing and storage of the attention module. Based on that, we propose `DistKV-LLM`, a distributed LLM serving system that dynamically manages KV Cache and effectively orchestrates all accessible GPU and CPU memories spanning across the data center. This ensures a high-performance LLM service on the cloud, adaptable to a broad range of context lengths. Validated in a cloud environment with 32 NVIDIA A100 GPUs in configurations from 2 to 32 instances, our system exhibited 1.03-2.4× end-to-end throughput improvements and supported context lengths 2-19× longer than current state-of-the-art LLM service systems, as evidenced by extensive testing across 18 datasets with context lengths up to 1,900K.

1 Introduction

Large Language Models (LLMs) [11, 14, 43]have fueled the rapid growth of LLM cloud services, becoming crucial infrastructure for advancing AI applications. However, this development faces significant challenges due to the massive computational and data requirements. These services typically use multiple GPU cards working together for LLM tasks. Yet, the dynamic nature of LLMs creates complex computational issues.

At the heart of LLM services lies the intrinsic procedure of auto-regressive text generation [13, 39, 42, 46], where the model generates one word (or token) at a time. Each newly generated token becomes appended to the existing text corpus, forming the input for recalibration within the LLM. This iterative progression persists until the ultimate word or token is generated. Crucially, the requisite memory and computational resources for LLM services dynamically oscillate throughout the LLM service, with neither the lifetime nor the length of the sequence known a priori.

The dynamic and iterative nature of auto-regressive text generation makes it impossible to plan resource allocation in advance [2, 26, 50], posing substantial challenges in designing efficient LLM service systems on the cloud. Particularly in long-context tasks, the expanding Key-Value (KV) Cache can surpass GPU memory limits within a computing instance, necessitating immediate resource reallocation. This often involves either initiating a costly live migration to transfer the task to a more capable instance or pre-assigning extra GPUs to handle potential memory overloads. The latter, however, can lead to inefficiencies and resource wastage, especially in tasks with normal length contexts.

Previous work, such as PagedAttention [26], has attempted to tackle these problems by facilitating the exchange (or swap) of data between GPU and CPU memory. However, this approach encounters several limitations. First, the scope of PagedAttention's memory swapping was restricted to the GPU and CPU memory *within* a single node, thus limiting its capacity to accommodate extremely long context lengths. Second, while its paging strategy is devised to minimize memory fragmentation, it swaps entire KV Caches on a request-level basis and thus missing the chance for more adaptive, granular scheduling in a distributed cloud environment. Last but not least, the interruption of computation for swapped-out requests can cause jittered performance for the running task, risking non-compliance with the strict service-level agree-

*: Equal contribution.

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

ments (SLAs) [36]that are crucial for cloud services.

To tackle the above challenges, we propose DistAttention, a novel attention algorithm designed to overcome these challenges. DistAttention partitions the KV cache into rBlocks—uniform sub-blocks that facilitate the distributed computation and memory management of attention modules for LLM service with long context length. Distinct from conventional methods that mainly utilize GPU or CPU memory within a single node, DistAttention enables optimization opportunities of all accessible GPU or CPU memory resources spread across the data center, particularly those that are now underutilized. This not only enables support for much longer context lengths but also avoids the performance fluctuations typically associated with data swapping or live migration processes.

In this paper, we developed DistKV-LLM , a distributed LLM service engine that integrates seamlessly with DistAttention. DistKV-LLM excels in managing KV Cache, efficiently coordinating memory usage among distributed GPUs and CPUs throughout the data center. When an LLM service instance faces a memory deficit due to KV Cache expansion, DistKV-LLM proactively seeks supplementary memory from less burdened instances. Moreover, DistKV-LLM introduces an intricate protocol that facilitates efficient, scalable, and coherent interactions among numerous LLM service instances running in the cloud. This protocol is designed to manage and balance the large amount of memory resources effectively. Additionally, DistKV-LLM prioritizes data locality and communication optimization, crucial for addressing performance challenges associated with the distributed storage of the KV Cache.

In summary, our work seeks to fully utilize all the available GPU resources across the data center, ensuring a smooth and efficient cloud service for LLMs especially when handling long-context tasks. The DistAttention, combined with DistKV-LLM, offers a solution to the resource allocation and optimization challenges faced by LLM services in distributed environments. This approach enables efficient resource management, allowing LLM services to handle a wide range of context-generation tasks effectively. We conducted a comprehensive evaluation of DistKV-LLM in a cloud setup equipped with 32 NVIDIA A100 GPUs, testing various distributed system configurations ranging from 2 to 32 instances. Our assessment included 18 benchmark datasets with context lengths extending up to 1,900K. Our system demonstrated 1.03-2.4 × end-to-end performance improvements over state-of-the-art work and a capability of supporting 2-19× longer context lengths.

This paper makes the following contributions:

- We present DistAttention , an innovative attention algorithm designed to significantly advance distributed computing for Large Language Models (LLMs) on the cloud. This algorithm is particularly adept at handling dynamic and diverse context generation tasks. Crucially, DistAttention unlocks the potential to fully utilize all available GPU and CPU memory resources across the data center.
- We introduce DistKV-LLM , a distributed LLM service engine, which excels in providing efficient, scalable, and coherent management of distributed KV Caches, harnessing the vast memory resources within GPU clusters on cloud infrastructure. DistKV-LLM also effectively optimizes memory locality and communication overhead, ensuring a smooth and efficient cloud service for LLMs.
- We demonstrate the feasibility and efficiency of the combination of DistAttention and DistKV-LLM in a cloud environment with 32 NVIDIA A100 GPUs and 18 datasets with up to 1,900K context length. Our system outperforms state-of-the-art work, delivering support for context lengths that are 2-19× longer and achieving 1.4-5.3 × higher throughput in tasks with standard-length contexts.

In the following sections of this paper, Section 2 introduces relevant background information. Section 3 outlines the key challenges of serving LLMs on the cloud and our main idea. Section 4 delves into the details of our design. Section 5 describes our implementation details. Section 6 presents our evaluation results. Section 7 provides an overview of related works. Section 8 concludes this work.

2 Background

2.1 Large Language Models

Transformer-based large language models (LLMs) have revolutionized natural language processing, offering capabilities ranging from simple text generation to complex problem-solving and conversational AI [15, 19, 20, 34].

2.1.1 Model Architecture

Large Language Models (LLMs) models [11, 14, 43] have a sophisticated architecture built on the principles of the Transformer model [46]. For example, GPT-3 [13], one of the largest models, consists of numerous transformer blocks (layers) with 175 billion parameters, enabling it to capture complex language patterns and generate human-like text. A Transformer block consists of several key components:

QKV Linear layer takes the input to the Transformer block first. These layers are essentially fully connected neural networks that project the input into three different spaces, including queries (Q), keys (K), and values (V).

Multi-Head Self-Attention Mechanism, or the attention module, is the core of the Transformer block. It allows the model to weigh the importance of different parts of the input sequence differently. In multi-head attention, the input is linearly transformed multiple times to form different 'heads', allowing the model to jointly attend to information from different representation subspaces at different positions.

Feed-Forward Neural Network, or FFN module, is after the self-attention module. This is typically a two-layer neural network with a ReLU activation in between. This network is identical for different positions but with different parameters from layer to layer.

2.1.2 LLM Service

Prefill Phase During inference, LLMs first receive a prompt or input text from the user. This input is processed to understand the context and the nature of the task required (e.g., answering a question, writing a poem, etc.).

Given a prompt of tokens $X = [x_1, x_2, \ldots, x_n]$ as the initial text, the model predicts the next token x_{n+1}. The probability distribution $P(x_{n+1}|X)$, representing the likelihood of each possible token being x_{n+1}, is computed as:

$$P(x_{n+1}|X) = \text{Softmax}(W \cdot h_n + b) \tag{1}$$

where W and b are the parameters learned from the final layer of the Transformer model, and h_n is the hidden state vector associated with the last word x_n.

Autoregressive Generation Phase In the auto-regressive phase, the model generates one word at a time, each new word being conditioned on the sequence of words generated so far. This phase is iterative and continues until the model produces a complete and coherent response or reaches a predefined limit (like a word count).

The autogressive generation phase starts with an initial context X_0, which could be an empty sequence or a prompt provided by the user. First, at each time step t, compute the probability distribution $P(x_t|X_{t-1})$ over the vocabulary for the next token x_t based on the sequence generated so far. Second, select the next word x_t with the highest probability from this distribution:

$$x_t = \text{argmax}(P(x_t|X_{t-1})) \tag{2}$$

Third, append the selected token x_t to the sequence to form a new sequence. This process repeats until a termination condition is met, such as the generation of an end-of-sequence token or reaching a maximum sequence length.

2.2 Parallelism Method for LLM Service

Large Language Models (LLMs) require substantial computational power and memory resource during serving or inference. To manage this, several parallelism strategies are employed to distribute the workload and accelerate processing.

2.2.1 Data parallelism

To handle the substantial volume of requests in cloud environments, data parallelism [49] is applied by replicating LLMs across the data center. The fundamental computational unit, termed an *instance*, has a copy of the full model. Each instance operates independently, processing distinct batches of requests concurrently.

Table 1: LLaMA2-13B, KV Cache size with context length

Context length	10k	100k	500k	1000k
KV Cache size	8.19GB	81.9GB	409.6GB	819.2GB
Misc size	26GB	26GB	26GB	26GB

Batching. In each instance, batching strategies are essential for improving throughput, allowing for the simultaneous processing of a greater number of requests. Due to the variability in the context length, requests usually have varying lifetime, requiring dynamic batching strategies. Various methods [18, 50], have been introduced to improve the throughput of LLM serving on GPUs.

2.2.2 Model Parallelism

Model parallelism is a technique used to accommodate the inference of LLMs that cannot fit entirely within the memory of a single GPU. It involves splitting the model across multiple devices or nodes. Model parallelism can be categorized mainly into two types: pipeline parallelism and tensor parallelism.

Pipeline parallelism. With pipeline parallelism, the layers of a model are sharded across multiple devices [22, 23, 32, 33]. It involves splitting the model into several stages or layers, each of which is processed on different computing units.

Tensor parallelism. It involves splitting the model's layers across multiple GPUs. For LLMs, tensor parallelism is crucial when individual layers of the model are too large for a single GPU. It allows large matrix operations within layers to be distributed across multiple GPUs. With tensor model parallelism, individual layers of the model are partitioned over multiple devices [41].

3 Motivation and Main Idea

There has been a notable surge in the evolution of long-context LLMs [11, 21, 30, 37], with the context window expanding from 2K [43] to an impressive 256K [37, 45] in merely a year. This progression continues unabated, with applications of LLMs now demanding support for even longer contexts.

This expansion in context length poses substantial challenges for LLM serving systems, particularly due to the escalating computational and memory requirements for KV Caches [38]. Table 1 depicts this trend, showing a steep escalation in KV Cache size that directly corresponds to the growing context length for the LLaMA2-13B model [44]. Current

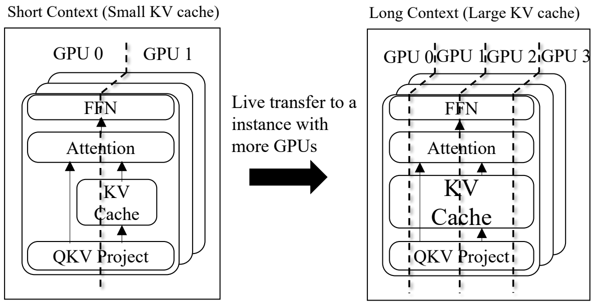


Figure 1: The dynamic and unpredictable resource demand of LLM service often requires either initiating a costly live migration to transfer the task to a more capable instance or pre-assigning extra GPUs to handle potential memory overloads.

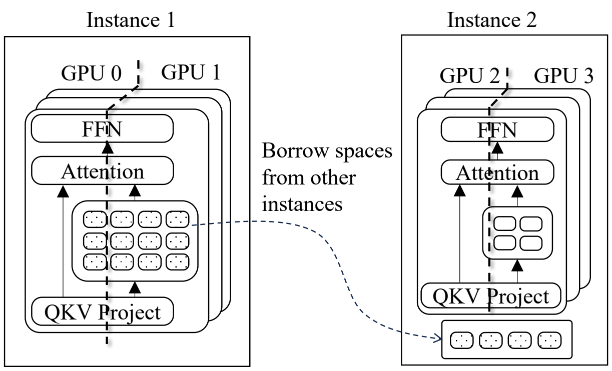


Figure 2: Our method enables KV Cache memory management in an elastic way, facilitating better performance and higher resource utilization in the cloud environment.

GPUs, with memory capacities spanning several dozen GBs, are being pushed to their limits, necessitating more memory space to accommodate the burgeoning size of KV Caches.

3.1 Challenges to LLM serving on Cloud

In this work, we endeavor to effectively utilize the vast memory capacities of GPUs and CPUs available in data centers, with the goal of creating an efficient memory pool specifically designed for LLM services capable of handling extremely long context lengths. However, the development of such a system is far from straightforward. We have identified two primary challenges that arise in the context of serving large language models with extended contexts on cloud platforms.

Challenge 1: significant disparities in memory demands obstacles efficient model parallelism. In stark contrast to the continuously expanding KV Cache throughout the auto-generation process, the memory requirements for the remaining activation tensors remain constant, as detailed in Table 1.

This disparity between the attention layer and other layers poses a substantial challenge for efficiently implement-

因为只有 attention layer 会产生 KV cache，
也因此会影响模型并行

KV cache 增长导致 GPU 数目需求增多，而其他 layers 的 tensor dimension 并不会随着 context length 增长，这就导致传统的 model parallelism 会让这些 layers 不必要地划分更细，进而导致资源利用率下降

ing model parallelism. To accommodate the extensive KV Cache necessary for long-context tasks, an increased number of GPUs is required. However, tensor dimensions in other layers do not scale with context length. As a result, traditional model parallelism leads to more fine-grained subdivisions of these layers when distributed across more GPUs, as shown in Figure 1, resulting in less efficient resource utilization.

Some previous studies [9, 26], have suggested dividing KV Caches into smaller blocks for more fine-grained memory management, aiming to prevent memory fragmentation. While these approaches have disaggregated the KV Cache of attention layers from the Transformer block, they are still reliant on gathering and positioning all blocks within the local GPU memory to carry out attention module's computation. In contrast, our design goal focuses on storing KV Caches and executing attention modules in a *distributed* manner, essential for effectively utilizing the abundant resources available in cloud environments. 关注 KV cache 的以分布式的方式进行存储和计算

Challenge 2: dynamicity of KV Cache size leads to inefficient resource management in the cloud environment. The intrinsic nature of the auto-regressive design determines that the ultimate sequence length remains unknown until the generation process reaches its conclusion, typically marked by an "ending" character. Consequently, memory requirements are completely dynamic and unpredictable, fluctuating significantly in scale. The demands can range from a few gigabytes to several hundred gigabytes, which is continuing to escalate even further.

This variability precludes any form of resource planning in advance. Resources must be allocated or released dynamically, reacting to the real-time demands of the auto-regressive process. If the memory required for a context exceeds the capacity of an instance's GPUs, the entire task must be transferred to a larger instance with more GPUs, a process known as live migration. Live migration is resource-intensive and, as our experiments show, can be 25x more costly than a standard inference. An alternative, allocating more GPUs to a computing instance from the outset, can lead to resource wastage for tasks involving shorter contexts, thereby compounding the challenge of efficient resource allocation.

PagedAttention [26] addresses the management of KV Caches by employing fine-grained sub-blocks, analogous to pages in operating systems. However, this approach is confined to utilizing only CPU memory for swapping, a method that proves inefficient on the cloud. The limitation imposed by the finite CPU memory not only restricts the maximum context length supportable by LLM services but also fails to capitalize on the expansive memory resources distributed across the cloud. In contrast, our objective is to harness the extensive memory capabilities of both GPUs and CPUs within data centers. We aim to establish an efficient memory pool, meticulously crafted for LLM services, to support the processing of exceptionally long context lengths effectively.

自回归过程的 seq_len 未知性导致我们必须动态地分配资源，包括 live migration 和 overprovision，因此本文建立一个 memory pool

3.2 Main Idea

Motivated by the above challenges, we present a suite of key techniques specifically designed to address these challenges. Together, they form a comprehensive and systematic approach, ensuring efficient LLM serving capable of handling extended context lengths.

To address challenge 1, we introduce a new attention algorithm named `DistAttention`. This algorithm breaks down the traditional attention computation into smaller, more manageable units known as macro-attentions (MAs) and their corresponding KV Caches (rBlocks). This innovative method facilitates the decoupling of KV Caches' computation from the standard transformer block, thereby enabling independent model parallelism strategies and memory management for attention layers versus other layers within the Transformer block. For non-attention layers, we apply established model parallelism strategies [17, 27, 41, 52]. In contrast, the attention layers are managed adaptively, dynamically allocating memory and computational resources across the data center in response to the fluctuations of the KV Cache.

To overcome challenges 2, we present `DistKV-LLM` , a distributed LLM service engine seamlessly integrated with `DistAttention`. The `DistKV-LLM` is designed to provide an efficient KV Cache management service, coordinating memory usage across GPUs and CPUs throughout the data center. When an LLM service instance encounters a memory shortfall due to an increase in the KV Cache, `DistKV-LLM` proactively identifies and borrows available memory spaces from other instances that have excess capacity, as is shown in Figure 2. This automated mechanism is realized by collaborative operations of two major components, the rManger and the gManager. The rManger virtualizes all the GPU and CPU memories within each LLM service instance , handling memory operation requests from both local and remote instances. Simultaneously, the gManager operates as a global coordinator, maintaining a protocol that ensures effective, scalable, and coherent resource management among distributed rManagers. Moreover, `DistKV-LLM` proposes a new algorithm, called DGFM, that effectively addresses the issue of memory fragmentation in the distributed KV Cache environment of the data center. This joint effort ensures continuous and efficient memory utilization, thereby enhancing the overall performance and reliability of the LLM service.

In summary, our integrated approach with `DistKV-LLM` and `DistAttention` presents a robust and scalable solution to the unique challenges posed by long-context LLM serving on the cloud. By addressing key issues related to memory management and computation scheduling, we ensure that LLM services can operate efficiently and adaptively in the cloud. This innovative framework not only optimizes resource utilization but also paves the way for future advancements in the field of large-scale language model deployment. We present details of our design in the following sections.

4 Method

4.1 Overview

In the following section, we begin by introducing `DistAttention`, an innovative attention algorithm crafted to facilitate distributed KV Cache management and computation, as detailed in Section 4.2. Based on this, we present `DistKV-LLM`, an LLM serving engine specifically designed for efficient KV caches management of distributed GPU memory at the cluster scale.

Our approach encompasses several key components: Firstly, in Section 4.3, we introduce the rManager, a software layer that virtualizes the GPU and CPU memories for each LLM service instance. It offers an abstraction layer for basic memory blocks, termed rBlocks, enhancing memory management efficiency. Secondly, we describe a comprehensive protocol-facilitated by the gManager, a global management system in Section 4.4. It ensures effective, secure, and coherent management of rBlocks across distributed GPUs in the data center. In Section 4.5, we further propose an innovative algorithm that is specifically designed to aggregate fragmented memory blocks, thereby significantly enhancing data locality within the distributed KV Cache system. Finally, in Section 4.6, we propose a series of optimization strategies designed to minimize the extensive communication overhead associated with the distributed storage of the KV cache, by effectively overlapping computation and communication tasks. Further details about this design are discussed in the following sections.

4.2 DistAttention

To tackle the complexities of memory management, we have developed a novel attention algorithm, `DistAttention`. This algorithm effectively dis-aggregates the KV cache into smaller, more manageable sub-blocks, thereby facilitating distributed memory management across the data center. Key to this approach is the partitioning of `DistAttention` into multiple Micro Attentions (MAs), with each MA encompassing a sub-sequence of KV cache tokens. The unique aspect of this design lies in its ability to compute the attention result by performing MAs separately. Upon completion of computations on their respective sub-blocks of token by all Micro Attentions (MAs), the final attention results are obtained through an aggregation process. This involves scaling and reducing the outputs of each MA. The methodology and precise formulation of this aggregation process are delineated in the following equations:

$$MA_{ij} = exp(Q_i {K_j}^T - max(Q_i {K_j}^T)) \tag{3}$$

$$Attention(Q,K,V) = Reduce(Scale([MA_{ij}]_{j=1}^{B_{kv}})) \tag{4}$$

where the *Reduce* is calculated as below:

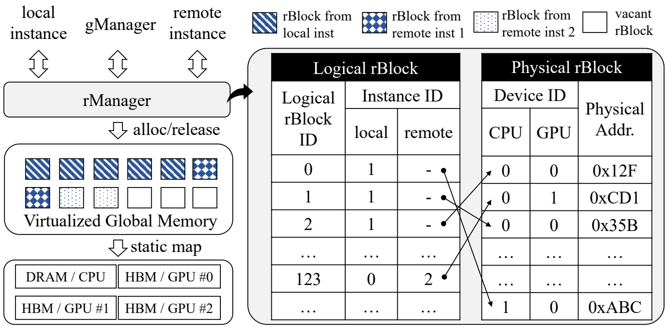


Figure 3: Illustration of the rManager design

$$
\begin{aligned}
&Reduce(Scale([MA_{ij}]_{j=1}^{B_{kv}})) \\
&= Reduce([exp(max(Q_iK_j^T) - max_i)MA_{ij}]_{j=1}^{B_{kv}}) \\
&= \sum_{j=1}^{B_{kv}} (exp(max(Q_iK_j^T) - max_i)MA_{ij}/sum_i)
\end{aligned} \tag{5}
$$

$$
max_i = max(max(Q_iK_1{}^T), \ldots, max(Q_iK_B{}^T))
$$

$$
sum_i = \sum (exp(Q_iK_j^T - max_i)
$$

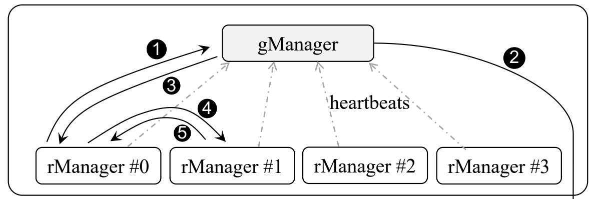
This approach not only consolidates the computations performed by individual MAs but also efficiently synthesizes them into a coherent final output, showcasing the effectiveness of the distributed processing framework implemented in our attention algorithm.

4.3 The rBlock and rManager

With the implementation of `DistAttention`, the Key-Value (KV) caches of LLMs are segmented into smaller units, known as rBlocks. Each rBlock encompasses a set of vectors corresponding to a fixed number of Key-Value tokens, along with essential metadata. This metadata provides critical information about the sub-block: the rBlock ID and Instance ID indicating the KV Cache in this rBlock whether belongs the local instance or a remote one; The device ID and physical ID labels the physical locations of this rBlock, which can be on the CPU side or one of the multiple GPUs.

Each LLM service instance is equipped with a dedicated rBlock manager, referred to as the rManager. The rManager is responsible for overseeing all the rBlocks located in local devices. It effectively virtualizes the global memory space of GPUs by dividing it into fixed-sized physical rBlocks. Each physical rBlock is designed to accommodate a single logical rBlock. The rManager maintains a detailed table that maps these logical rBlocks to their corresponding physical rBlock addresses in the global memory, as is shown in Figure 3.

The rManager offers a unified API interface to serve both local and remote memory operations. These operations include allocating physical rBlocks for the newly generated KV caches and releasing them when no longer needed. Upon receiving a memory allocation request, either from a local or a remote instance, the rManager consults the rBlock table to identify the first available physical rBlock space. In scenarios where sufficient space is *unavailable*, the rManager initiates a space-borrowing procedure from other instances. More details will be elaborated in Section 4.4. Notably, if the allocation request originates from a remote instance, the rManager is programmed to automatically return a false response, indicating the unavailability of direct remote allocation. We apply a cap on the number of rBlocks that can be allocated to remote instances, which is determined experimentally and configured as a hyper-parameter.



Global Debt Ledger

Inst. ID	Un-used/Total Mem.	Debtor 0		Debtor 1	
		Inst #	Mem.	Inst #	Mem.
0	30/80 (GB)	1	7 (GB)	3	5 (GB)
1	8/80 (GB)	-	-	-	-
2	10/80 (GB)	-	-	-	-
3	16/80 (GB)	1	11 (GB)	-	-
…	…	…	…	…	…

Figure 4: gManager and Contract Protocol. The global debt ledger is a core component of the gManager, tracking each instance's memory usage. This table includes details about available spaces and spaces lent to its debtors. It outlines five instances, each illustrating different memory usage dynamics. Inst-0, with a relatively light workload, is to lend spaces to Inst-1&3. Inst-1, dealing with a long context, is borrowing space from both Inst-0&3. Inst-2 neither borrows nor lends. Inst-3, finds itself both borrowing (from Inst-0) and lending (to Inst-1) simultaneously, exemplifying a scenario in Section 4.5.

4.4 The gManager and Contract Protocol

The key component of our system, termed the *gManager*, functions as a centralized manager, maintaining the global memory information across all instances. Each instance periodically transmits heartbeat signals to the gManager, conveying updates about their remaining available memory space. Utilizing this data, the gManager constructs a detailed table known as the *global debt ledger*, as is shown in Figure 4.

Whenever an instance runs short of its memory space for rBlocks, the corresponding rManager seeks to borrow GPU or CPU memory spaces from neighboring instances. Prior to this,

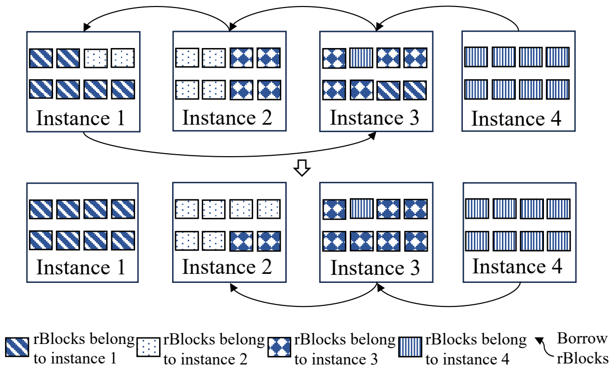


Figure 5: An illustration of our algorithm for fragmented memory management, where we conceptualize the problem as a search and elimination of circles within a directed graph.

Algorithm 1: Find a Circle in a Directed Graph

```
Data: Directed graph G with nodes and directed edges
Result: A circle in the graph G if found, otherwise false
Function HasCycle(G, v, visited, parent):
    visited[v] ← True;
    foreach neighbor u of v in G do
        if ¬ visited[u] then
            if DFS(G, u, visited, v) then
                return True;
        else if u ≠ parent then
            return True;
    return False;
Main:
    foreach node v in G do
        Initialize an array visited with all nodes set to False;
        if HasCycle(G, v, visited, −1) then
            return The circle found starting from node v;
    return False;
```

the rManager, acting as a *debtor*, is required to initiate a query to the gManager ❶, telling the size of the memory space it needs to borrow. Upon receiving this query, the gManager consults the global debt ledger ❷ and responds by providing potential *creditor's* address IDs ❸, which represent instances that currently have surplus memory space. The selection process adheres to a **locality & availability** principle, whereby the creditor instance is chosen based on the lowest relative communication cost and the highest available memory space. The gManager proposes three potential creditor IDs as recommendations. Subsequently, the debtor instance approaches these creditor instances sequentially with requests ❹, continuing until successful allocation is confirmed by one of them ❺. In cases where all three candidates return a negative response, the debtor instance will revert to the gManager for alternative suggestions. This dynamic and responsive system ensures efficient and effective memory space allocation and management across the data center.

In the following paragraphs, we describe the key components and design considerations of the contract protocol.

Global Debt Ledger : The *global debt ledger*, managed by the gManager, is a crucial table that chronicles the available memory and inter-instance debts across the network. Each entry in this ledger represents an LLM service instance, detailing the instance ID alongside its available memory spaces. Subsequent sub-fields in each entry denote the IDs of debtor instances, along with the quantity of rBlocks they have borrowed from their respective creditors.

Competing Candidates : In scenarios where multiple debtor instances concurrently send requests to a rManager, the system must navigate these competing demands efficiently. The *global debt ledger* plays an important role here, enabling the gManager to evenly distribute requests among instances, thereby preventing an overload on any single instance. On the other side, the rManager adopts a *first-come-first-serve* policy for allocating physical spaces to rBlocks from remote instances. If the rManager finds itself unable to allocate sufficient physical rBlocks for remote rBlocks due to space constraints, it responds with a false to the debtor instances. This response also prompts the gManager to update its records of the current resource availability, effectively pausing the forwarding of new requests until more resources become available. This approach ensures a balanced and orderly allocation of memory resources, mitigating potential bottlenecks in the system.

Coherency : We employ a *loose* coherence policy between the gManager and the rManagers. Under this approach, the gManager is not required to meticulously track every memory allocation or release action across all instances. Instead, it gathers this information through regular heartbeats that are automatically sent by the rManagers. Consequently, the gManager maintains an overview of general space usage throughout the data center rather than detailed, real-time data. When responding to a debtor rManager's request for borrowing space, the gManager only provides recommendations of potential creditor candidates. The debtor then must engage in negotiations with these suggested creditors to finalize the memory allocation. Situations involving multiple concurrent requests to the same rManager are managed using the previously discussed competing candidate strategy. This loosely coupled coherence framework not only streamlines operations but also minimizes excessive transaction overheads, thereby reducing processing latency and enhancing overall system performance.

Scalability : To meet varying throughput demands, the gManager is designed to enhance scalability through the deployment of multiple processes that concurrently handle querying requests. To expedite the process of identifying instances with surplus memory, the gManager periodically initiates a sorting operation. This operation arranges the instances based on their remaining available memory space, enabling querying requests to efficiently bypass instances with minimal memory resources. This approach ensures that the gManager operates within its optimal capacity, maintaining system efficiency and responsiveness while scaling to accommodate the dynamic needs of the network.

4.5 Fragmented Memory Management

Due to the dynamicity in variable context length and batching, a critical challenge emerges in the form of fragmented memory management[1]. Each instance within the system operates both as a creditor and a debtor of memory space, lending to and borrowing from other instances as required. For example, instances handling requests with long contexts may continuously grow, necessitating borrowing space from remote instances. Conversely, instances with short-lived requests release memory space sooner, which can then be lent to others or allocated to new requests. This dynamicity leads to a significant issue: *the deterioration of data locality*. As instances frequently access data stored in remote memory locations, the system incurs a substantial performance penalty, such as increased latency and reduced throughput.

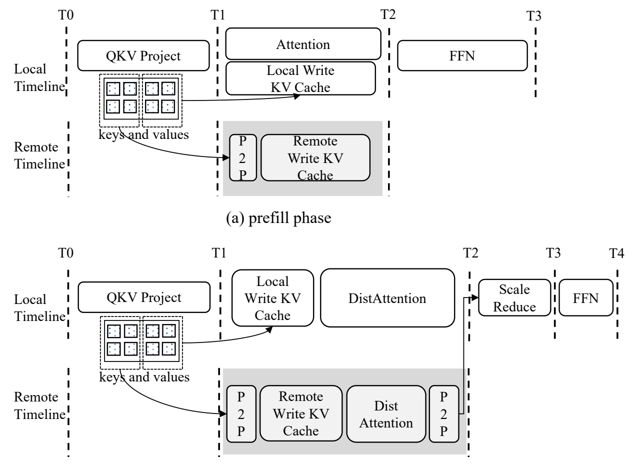
We propose a debt-graph-based fragmented memory management algorithm, namely DGFM, which aims to counteract this by strategically recalling memory spaces that have been lent out and swapping them for local data storage. A key challenge to this problem is that a large number of LLM service instances run concurrently in the data center, often involved in intricate debt relationships. To effectively manage this complexity, we conceptualize the problem as a search for circles within a directed graph. Initially, we construct a directed graph mirroring these debt relationships, where each node symbolizes an instance and every directed edge signifies the debt owed from one instance to another. Our algorithm is then applied iteratively. During each iteration, the algorithm selects a node at random and traverses the graph to identify a directed circle. The discovery of such a circle is pivotal; it indicates that the involved nodes, or instances, can mutually resolve their debts. This resolution is achieved through a strategic recall and swap of their respective memory blocks, thus enhancing overall system efficiency and memory utilization. Details of this algorithm is shown in Figure 5 and Algorithm 1.

通过在有向图里查找并消除环来让 instances 自行缓解碎片化

This directed graph is derived from the global debt ledger, and the DGFM algorithm is executed by the gManager. When a directed cycle is identified, the gManager issues requests to the rManager in the corresponding instances, freezing them from modifications or cancellations. We set an empirical threshold for the minimum number of memory blocks (rBlocks) to be swapped at each node, preventing inefficient recall and swap operations on overly small memory blocks. This process significantly reduces the need for remote memory access, thereby enhancing data locality, and ultimately leading to a noticeable improvement in system performance.

不去 swap 过小的 mem 而是直接在消除环之后直接分配在 local GPU memory?

[1]This memory fragmentation is particularly pertinent to distributed KV cache management in the context of LLM serving on the cloud. To address fragmentation concerns *within* the instance, we incorporate strategies from previous research [26].



(a) prefill phase

(b) auto-regressive phase

Figure 6: Dataflow of the overlapped computation and communication in prefill phase and auto-regressive phase.

4.6 Communication Optimization

Distributed KV Cache storage faces another challenge: the communication overhead of transferring rBlocks back and forth. During the execution of long-context tasks in LLM services, both the prefill and auto-regression phases can generate a substantial amount of KV Cache, incurring the rManager borrowing remote spaces. We have made specific optimizations for both scenarios, as is shown in Figure 6.

During the prefill phase, the memory demands of the KV Cache can be precisely predicted based on the prompt's length. This foresight enables pre-planned allocation of rBlocks—designated as either local or remote depending on their storage location. When executing the attention layers of the Transformer block, we overlap the computation of attention with the transfer of remote rBlocks.

In the auto-regression phase, rBlocks' allocation are handled dynamically. Simply repatriating all rBlocks for local computation incurs excessive network traffic. Moreover, given that the attention module's computation is fundamentally a vector-matrix multiplication—a notably memory-intensive task—localizing all computations can severely degrade system performance. The innovation of `DistAttention` allows us to redirect query vectors to the instance containing the remote rBlocks, facilitating the macro-attention computations there before sending the results back for integration. This approach significantly reduces data transfer volume by a factor of N, with N representing the count of tokens in the KV cache. A limitation of this method is its potential to vie for computational resources with the host instance of the remote rBlocks. To mitigate this, a threshold is established within

因为自回归时 QKV 的计算主要是 GEMV，将分散的 rBlocks 传到 local 进行计算会带来 memory bottleneck，因此可以利用 DistAttention 的 MA 来进行分布式计算，再聚合结果

each rManager, which adjudicates the borrowing of computational resources in accordance with local SLA guidelines, thereby ensuring a balanced and judicious use of the system's computational assets.

5 Implementation Details

`DistAttention` contains two types of operators, namely DistAttn and ScaleReduce, developed with approximately 5,000 lines of C++/CUDA code. The DistAttn operator is designed for distributed attention computation, with the results consolidated by the ScaleReduce operator to yield the final outcome. To adeptly manage a wide range of input context lengths, DistAttn incorporates an adaptive kernel selection process based on the dimensions of the inputs. Context lengths are categorized into three groups: normal range (0-8k), long range (8k-32k), and ultra-long range (>32k), for which we have developed and meticulously optimized three distinct kernel templates. Additionally, we have devised and implemented a heuristic approach to fine-tune CUDA kernels for specific tensor shapes.

On the other hand, `DistKV-LLM` adapts the Ray framework [31] to establish a distributed KV Cache management and scheduling system, developed with around 12,000 lines of Python code. For effective implementation of requests and network data movements, we customize the package encodings and transfers data packages with socket, instead of using RPC based framework. To maximize the high bandwidth benefits of RDMA [25], NCCL [1] is employed for cross-instance GPU tensor communication, ensuring efficient data exchange among distributed instances.

6 Evaluation

In this section, we present the evaluation results of our work.

Environment. We deploy `DistKV-LLM` on a cluster with 4 nodes and 32 GPUs. Each node has 8 NVIDIA A100 (80GB) GPUs.

Models. Our framework can now support most of popular LLMs such as GPT [13, 35], LLaMA [44], BLOOM [47] etc. Since most LLM models have similar backbone Transformer block, we choose one representative model, LLaMA2 [44] for evaluation. 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) [46], while the 70B model employs Grouped-Query Attention (GQA) [40].

Baseline. We select vLLM [26], the state-of-the-art LLM serving engine, as the primary baseline. Moreover, most previous LLM service systems use tensor parallelism. To validate the pipeline parallelism with contiguous batching, we implement similar design in Alpa [52] in vLLM framework as one of the baselines.

Datasets. We evaluate our system with 18 datasets, categorized into three types based on context length distributions. Each dataset comprises 1,000 text-generation tasks, derived from scenarios encountered in real-world applications. As is listed in Table 2, these datasets feature context lengths varying from 1 to 1,900K, with the proportion of long context tasks ranging from 1% to 30%.

Table 2: Datasets for Different Scenarios

Model, GPUs	Dataset IDs	Normal Request Range	Long Request Range	Long Request Ratio (%)
7B, 2	1, 7, 13	1-100k	100k-200k	1, 10, 30
13B, 4	2, 8, 14	1-140k	140k-280k	1, 10, 30
70B, 8	3, 9, 15	1-300k	300k-600k	1, 10, 30
13B, 8	4, 10, 16	1-240k	240k-480k	1, 10, 30
7B, 16	5, 11, 17	1-600k	600k-1200k	1, 10, 30
7B, 32	6, 12, 18	1-950k	950k-1900k	1, 10, 30

6.1 Context Length Benchmark

We evaluate and compare `DistKV-LLM` and the baseline's performance on different context lengths. We evaluate on three models with different context ranges. For LLaMA2-7B, we evaluate the task of 1-200k on 2 GPUs, 1-1200k on 16 GPUs, and 1-1900k on 32 GPUs respectively. For the LLaMA2-13B model, we tested 1-280k on 4 GPUs, and 1-480k on 8 GPUs respectively. For the LLaMA2-70B model, we tested range 1-450k on 8 GPUs.

To validate the performance of `DistKV-LLM` , we compare with two vLLM baseline versions. vLLM-v1 contains the same number of GPUs as `DistKV-LLM` in a single instance. Figure 7 shows the throughput of vLLM-v1, vLLM-v2 and `DistKV-LLM` across varing context length. Notably, `DistKV-LLM`(blue) not only achieves a throughput comparable to vLLM-v1 (green) but also supports substantially longer context lengths, approximately 2x-19x as demonstrated in Figure 7. This improvement is attributed to `DistKV-LLM`'s ability to efficiently coordinate memory usage across all instances, while vLLM-v1 is limited to the instance's private memory.

vLLM-v2 is pre-assigned with more GPUs so that it can support comparable context length with `DistKV-LLM`. By comparing with vLLM-v2(red), we demonstrate that `DistKV-LLM` sustains similar extended context lengths but achieves significantly higher throughput. As is shown in Figure 7, `DistKV-LLM` achieves 1.4x-5.3x higher throughput than vLLM-v2. This is because `DistKV-LLM` can maintain an efficient model parallelism strategy while vLLM-v2 partitioning the model into smaller segments across more GPUs, which results in lower hardware efficiency.

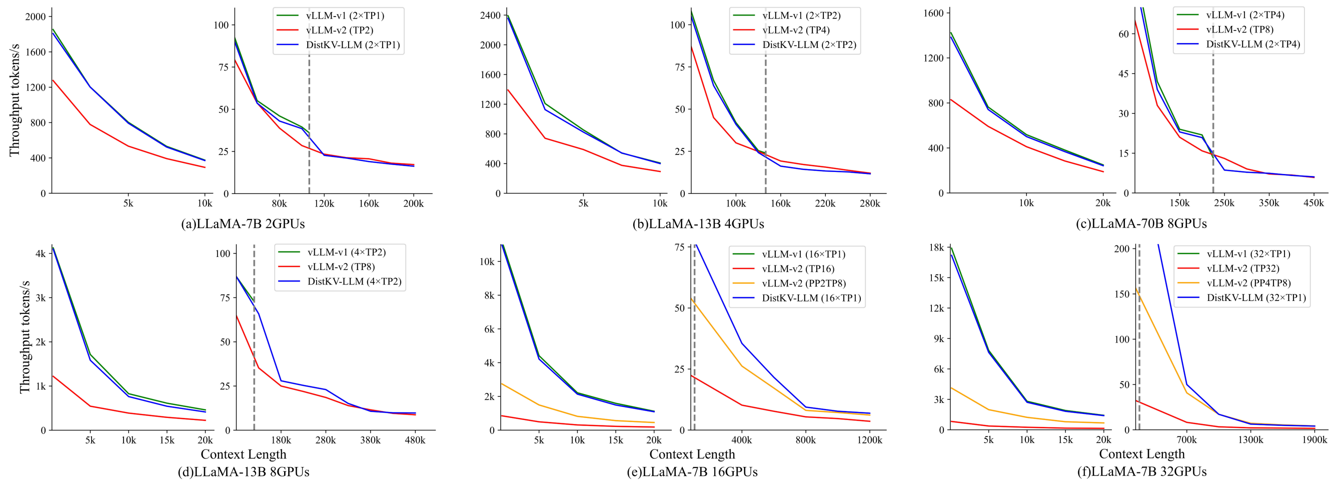


Figure 7: Throughput of a largest batch of requests with same specified context length.

6.2 End-to-end Serving Performance

We adopted the experimental setup from subsection 6.1, running the corresponding context range datasets to evaluate the end-to-end performance of the DistKV-LLM . The experiment result is shown in Figure 8. When the curve rises sharply, it indicates that the throughput has reached the system's limit, requests begin to queue up, and latency increases rapidly.

In the dataset with 1% long requests, DistKV-LLM achieves an improvement of approximately 1.4x to 2.4x over the baseline. This is because splitting the model into smaller fragments leads to lower GPU utilization, which considerably reduces the efficiency of linear computations. In the dataset with 10% long requests, DistKV-LLM achieves a performance improvement of approximately 1.1x to 1.4x compared to the baseline. In a dataset where long requests comprise 30% of the data, DistKV-LLM realizes a performance gain of about 1.03x to 1.3x over the baseline. As the proportion of long requests in the dataset increases, the performance gain offered by DistKV-LLM diminishes. This is because when the model processes requests with long context, there is a lower ratio of linear computations to attention computations. The performance gain that DistKV-LLM has in the linear component becomes a smaller fraction of the overall computational workload, and the attention component's performance does not show a significant advantage over the baseline.

6.3 Live Migration

An alternative solution to the varying context length is live migration, which makes an on-demand transfer to a more capable instance with more GPUs. In this experiment, we compare DistKV-LLM and live migration on LLaMA2-7B model. For the new instance, LLM model is downloaded through the Amazon Simple Storage Service (Amazon S3) [3] and loaded by vLLM in the format of SafeTensor [7].

Initially, we deployed the service using an A100 GPU, which can handle requests up to a maximum length of 108k. When the context length exceeds 108k, an additional A100 GPU should be utilized for expansion. The result is shown in Figure 9. The horizontal axis represents the length of the prompt and the length of the output. The vertical axis indicates the latency of generating the corresponding output length. The overhead caused by the live migration is 45x that of the communication overhead in DistKV-LLM. When the context length is 105k prompt and 5k output, it triggers a live migration. In this scenario, the latency of vLLM significantly increases, whereas DistKV-LLM only experiences a negligible disturbance. When generating tokens of lengths 5k, 10k, 20k, and 30k, the completion time of DistKV-LLM is respectively 3.5×, 2.2×, 1.6×, and 1.4× faster than that of vLLM. This is because migrating the entire service results in substantial overhead, which includes remotely downloading the model (despite utilizing high-speed download links) and the inference engine loading the model. In contrast, DistKV-LLM merely needs to establish a connection with the expanded devices without the need for downloading and loading the model.

6.4 Optimizing Memory Allocation

The dynamicity in variable context length and batching leads to the deterioration of data locality. The DGFM algorithm aims to optimize memory allocation by recalling lent memory spaces. In this experiment, we deployed a service using DistKV-LLM with four LLaMA2-13B tp2 instances, capable of handling request lengths ranging from 1 to 480k. We compared the throughput performance of the service with DGFM enabled and without DGFM, and the results are depicted in the Figure 10. In the initial phase, the performance of services with DGFM enabled and disabled is similar. Over time,

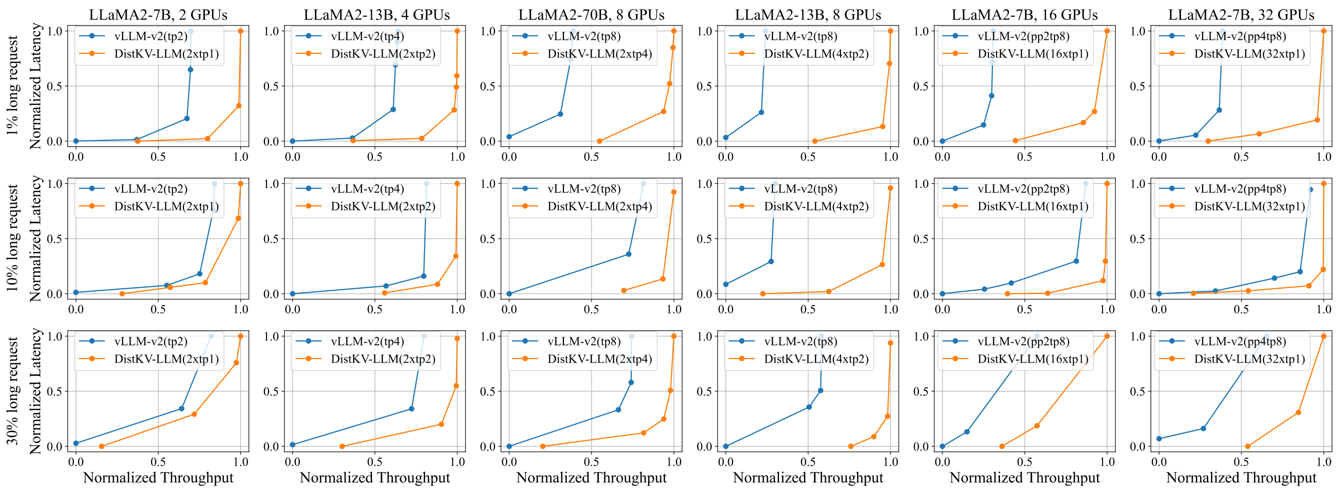


Figure 8: End-to-end serving performance

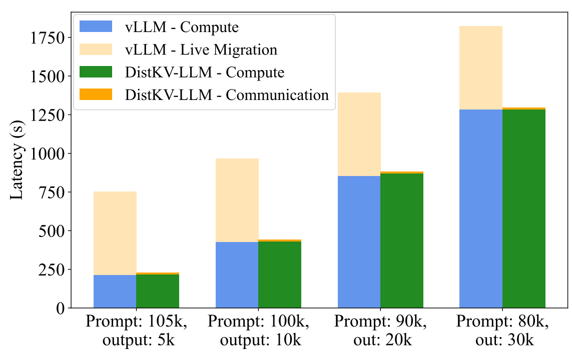


Figure 9: Comparison of live migration overhead.

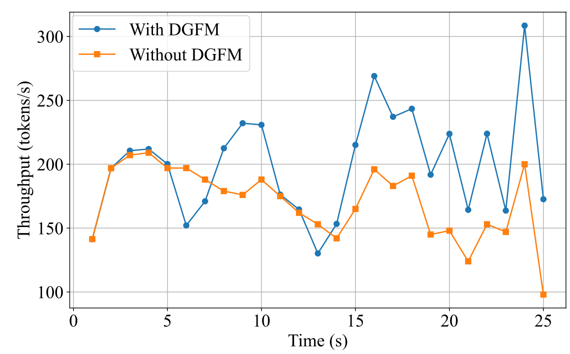


Figure 10: Throughput Over Time with and without DGFM Enabled.

the data locality issue begins to emerge. Services with DGFM maintain a higher overall throughput by periodically clearing the debt circle and optimizing the overall data locality in the distributed environments. In contrast, services without DGFM experience an increasingly severe problem with data locality deterioration, resulting in a continuous downward trend in overall throughput.

6.5 Ablation Study

`DistAttention` Breakdown. `DistAttention` introduces the capability for distributed storage and computation of attention across instances, which also incurs certain overheads. These are primarily due to the transmission costs of the query, key, and value tensors for individual tokens, as well as some other overheads such as tensor slicing and concatenating. We deployed two instances on 8xA100 GPUs, which share storage and computational resources through `DistKV-LLM`. We conducted a breakdown analysis of the runtime of `DistAttention` and compared it with the attention that is divided into eight parts by Tensor parallelism. The result is shown in Figure 11. Compared to TP8 Attention, the additional overhead introduces a 5%-10% increase in latency. This extra latency is almost constant, which means that as the context length increases, the proportion of this overhead becomes increasingly smaller.

Comparing Remote Compute and Local Compute. There are two strategies to compute the remotely allocated rBlocks, 1) local compute: bring back the KV Cache from the remote instance via a high-speed interconnect network to perform the full attention computation locally; 2) remote compute: transmit the query vector to the remote instance, where the rBlocks locate, to carry out distributed attention computations, enabled by `DistAttention`, and then retrieve the result vector back. The comparative results of these two methods are illustrated in the Figure 12. The latency of local compute is significantly higher than that of remote compute, which is attributed to

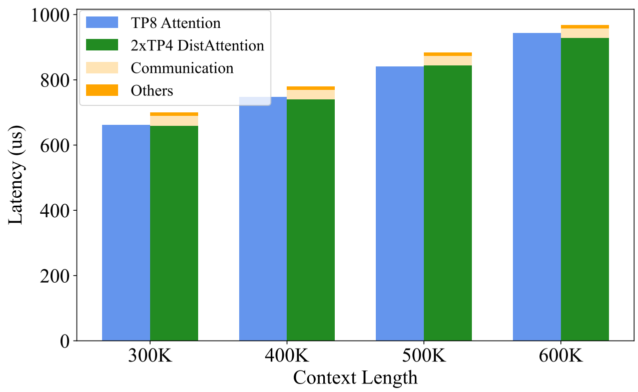


Figure 11: Attention breakdown with different context lengths.

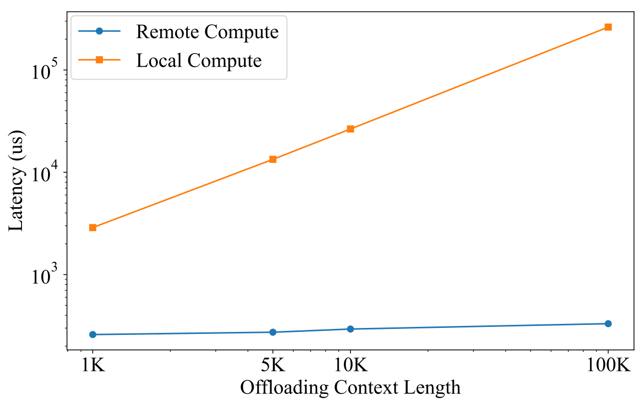


Figure 12: Comparison between Local Compute and Remote Compute.

the fact that local compute requires transferring a large volume of remote KV Cache back to the local instance through the network, constrained by network bandwidth, substantially increasing the overall latency. In `DistKV-LLM`, we use this experiment results in the rManger to guide the communication optimizations discussed in Section 4.6.

7 Related Works

Existing LLM service systems. Numerous LLM serving systems have been proposed recently. ORCA [50] has introduced an iteration-level scheduling strategy which greatly enhances the computation and memory utilization in batching inference. To address the issue of memory wastage resulting from fragmentation and redundant replication, vLLM [26] has developed a Paged KV (Key-Value) Cache and Paged Attention mechanism. DeepSpeed-FastGen [4] has proposed a novel prompt and generation composition strategy called Dynamic SplitFuse, which is designed to further enhance continuous batching and system throughput. AlpaServe [27] explores the opportunity of statistical multiplexing by model parallelism in the scenario of bursty request rate. FasterTransformer [2] and DeepSpeed Inference [10] have implemented pioneered and extensive kernel-level performance optimizations specifically for Transformer models. TGI [5], TensorRT-LLM [8] and lmdeploy [6], building upon FasterTransformer, have adapted features like Contiguous Batching and Paged Attention. 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.

Comparison to Ring Attention. Ring Attention [28,29] was introduced as a method to distribute long sequences across multiple devices, with the intent of fully overlapping the communication of key-value (KV) blocks with the computation of blockwise attention. This approach is highly efficient for training with long sequences and for the prefill phase during inference. However, when it comes to the decoding phase in inference, the transfer of KV blocks between devices cannot be concealed by computation leading to substantial overhead. Figure 12 depicts the overhead of transferring KV blocks is significantly higher than communication of `DistAttention` .

Solutions for Long Context Serving. Another category of methods to address the challenge of managing oversized Key-Value (KV) Cache for long-context inference involves sparse KV Caches, such as Sliding Window Attention [12, 16, 24]. This technique only requires maintaining a KV Cache the size of the window. Both H2O [51] and StreamingLLM [48] also retain a fixed window size for the KV Cache, but they mitigate the precision loss due to context information discarding by employing a KV cache eviction algorithm and incorporating an Attention Sink, respectively. However, since these methods discard some context information, they inevitably compromise the effectiveness of Large Language Models (LLMs) to some extent.

8 Conclusion

The dynamic, auto-regressive nature of LLM inference computation poses significant challenges to LLM service on the cloud, especially for tasks with long-context sequences. Addressing these challenges, we introduce `DistAttention`, an innovative distributed attention algorithm that efficiently segments the KV cache into manageable units for distributed processing. Complementing this, `DistKV-LLM`, a distributed LLM service engine which excels in KV Cache management, optimally coordinates memory usage across the data center. This combination of `DistAttention` and `DistKV-LLM` effectively ensures a smooth and efficient cloud service for LLMs especially when handling long-context tasks. In a comprehensive evaluation using 32 NVIDIA A100 GPUs and 18 datasets, our system showed 1.03-2.4 $\times$ performance improvements over existing solutions and supported context lengths 2-19 $\times$ longer, demonstrating its effectiveness in managing a wide range of context-generation tasks.

References

[1] Nvidia collective communication library. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/index.html, 2020.

[2] Fastertransformer. https://github.com/NVIDIA/FasterTransformer, 2021.

[3] Amazon s3: Object storage built to retrieve any amount of data from anywhere. https://aws.amazon.com/s3, 2023.

[4] Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference. https://github.com/microsoft/DeepSpeed/tree/master/blogs/deepspeed-fastgen, 2023.

[5] Large language model text generation inference. https://huggingface.co/docs/text-generation-inference, 2023.

[6] Lmdeploy. https://github.com/InternLM/lmdeploy, 2023.

[7] Simple, safe way to store and distribute tensors. https://huggingface.co/docs/safetensors, 2023.

[8] Tensorrt-llm. https://github.com/NVIDIA/TensorRT-LLM, 2023.

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

[10] 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.

[11] 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.

[12] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. arXiv preprint arXiv:2004.05150, 2020.

[13] 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.

[14] 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.

[15] 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.

[16] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers, 2019.

[17] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. Dapple: A pipelined data parallel approach for training large models. In Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 431–445, 2021.

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

[19] Github. https://github.com/features/copilot, 2022.

[20] Google. https://bard.google.com, 2023.

[21] Chi Han, Qifan Wang, Wenhan Xiong, Yu Chen, Heng Ji, and Sinong Wang. Lm-infinite: Simple on-the-fly length generalization for large language models. arXiv preprint arXiv:2308.16137, 2023.

[22] 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.

[23] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. Proceedings of Machine Learning and Systems, 1:1–13, 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] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using rdma efficiently for key-value services. ACM SIGCOMM Computer Communication Review, 44:295 – 306, 2014.

[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] 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.

[28] Hao Liu and Pieter Abbeel. Blockwise parallel transformer for long context large models. arXiv preprint arXiv:2305.19370, 2023.

[29] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context. arXiv preprint arXiv:2310.01889, 2023.

[30] Rui Mao, Guanyi Chen, Xulang Zhang, Frank Guerin, and Erik Cambria. Gpteval: A survey on assessments of chatgpt and gpt-4. arXiv preprint arXiv:2308.12488, 2023.

[31] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In 13th USENIX symposium on operating systems design and implementation (OSDI 18), pages 561–577, 2018.

[32] 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.

[33] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel dnn training. In International Conference on Machine Learning, pages 7937–7947. PMLR, 2021.

[34] OpenAI. https://openai.com/blog/chatgpt, 2022.

[35] OpenAI. Gpt-4 technical report, 2023.

[36] Pankesh Patel, Ajith H Ranabahu, and Amit P Sheth. Service level agreement in cloud computing. 2009.

[37] Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. Yarn: Efficient context window extension of large language models, 2023.

[38] 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.

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

[40] Noam Shazeer. Fast transformer decoding: One write-head is all you need. arXiv preprint arXiv:1911.02150, 2019.

[41] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[42] 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.

[43] 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.

[44] 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 fine-tuned chat models. arXiv preprint arXiv:2307.09288, 2023.

[45] Szymon Tworkowski, Konrad Staniszewski, Mikołaj Pacek, Yuhuai Wu, Henryk Michalewski, and Piotr Miłoś. Focused transformer: Contrastive training for context scaling. arXiv preprint arXiv:2307.03170, 2023.

[46] 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.

[47] 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 176b-parameter open-access multilingual language model. arXiv preprint arXiv:2211.05100, 2022.

[48] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. arXiv preprint arXiv:2309.17453, 2023.

[49] Eric P Xing, Qirong Ho, Wei Dai, Jin-Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. Petuum: A new platform for distributed machine learning on big data. In Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pages 1335–1344, 2015.

[50] 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.

[51] 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. H_2o: Heavy-hitter oracle for efficient generative inference of large language models, 2023.

[52] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), pages 559–578, 2022.