KunServe: Elastic and Efficient Large Language Model Serving with Parameter-centric Memory Management

Rongxin Cheng, Yifan Peng, Yuxin Lai[†], Xingda Wei[*], Rong Chen, and Haibo Chen

Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University

Abstract

The stateful nature of large language model (LLM) serving can easily throttle precious GPU memory under load burst or long-generation requests like chain-of-thought reasoning, causing latency spikes due to queuing incoming requests. However, state-of-the-art KVCache centric approaches handle load spikes by dropping, migrating, or swapping KVCache, which faces an essential tradeoff between the performance of ongoing vs. incoming requests and thus still severely violates SLO.

This paper makes a key observation such that model parameters are independent of the requests and are replicated across GPUs, and thus proposes a parameter-centric approach by selectively dropping replicated parameters to leave precious memory for requests. However, LLM requires KVCache to be saved in bound with model parameters and thus dropping parameters can cause either huge computation waste or long network delay, affecting all ongoing requests. Based on the observation that attention operators can be decoupled from other operators, this paper further proposes a novel remote attention mechanism through pipeline parallelism so as to serve upcoming requests with the additional memory borrowed from parameters on remote GPUs. This paper further addresses several other challenges including lively exchanging KVCache with incomplete parameters, generating an appropriate plan that balances memory requirements with cooperative execution overhead, and seamlessly restoring parameters when the throttling has gone. Evaluations show that KUNSERVE reduces the tail TTFT of requests under throttling by up to $27.3\times$ compared to the state-of-the-art.

1 Introduction

Transformer-based large language models (LLMs) are reshaping the computing industry. Such models generate output in a streaming fashion, producing results token by token. The tokens are used by downstream tasks like chatbots [34], copilots like programming assistants [23], and interactive agents [20]. These tasks commonly involve human interactions, which are sensitive to latency above 1s [11, 56], and shorter is better even within 1s [22]. Thus, both the time to generate the first token (TTFT) and the time between subsequent tokens (TBT)

are important to user experiences.

Compared to prior AI models like vision models, LLM serving is *stateful*: before generating the final token, the intermediate results (called KVCache) must be kept in the GPU memory (i.e., high-bandwidth memory, HBM). Such a stateful generation pattern presents a key challenge: the serving latency (especially TTFT) could spike (up to $190\times$ in BurstGPT [49] (§2.2) and others (§5)) when precious HBM experiences throttling during spikes in KVCache requirements. Such throttling is common: when the incoming request rate spikes, which is commonly found in real-world workloads [36, 18], the GPU memory demands also spike because they are proportional to the number of requests. Besides, even a small number of requests can cause throttling when they generate many tokens, since KVCache demand grows with the token count. This could happen because modern LLM workloads increasingly generating more tokens to achieve better performance. A concrete example is the chain-of-thought reasoning [3]: with more tokens generated (either explicitly by prompting [50] or implicitly via training [54]), the LLMs have been proven to show enhanced problem-solving abilities [31].

Memory throttling severely impacts model serving latency, because requests must wait for GPUs to free up sufficient memory for processing. Unfortunately, LLMs typically take seconds to generate all tokens and release memory. Hence, if throttling occurs, the incoming requests must be queued for such long so as to proceed, resulting in latency spikes. This paper answers a key question: *how can we effectively handle the latency spikes caused by memory throttling during LLM serving?*

State-of-the-art approaches use a *KVCache-centric approach* to handle memory throttling [29, 51, 39, 45]. When a GPU lacks sufficient HBM for incoming requests, the system adjusts the KVCache for active requests—dropping it, swapping it out, or migrating it to an available spare GPU to make room for waiting requests (detailed in§2.3). However, KVCache-centric memory management still has a lot of deficiency for handling latency spikes. (with tail latencies still 83–126 × higher than normal, detailed in §2.3). Since KVCache is essential to LLM inference, KVCache-centric approaches face an essential tradeoff between the performance of ongoing and incoming requests: pruning a large amount of KVCache significantly disrupts ongoing requests, while pruning only small amounts provides little memory for incoming

---

[*]Xingda Wei is the corresponding author (wxdwfc@sjtu.edu.cn)
[†]Work done during an internship at SJTU. Yuxin Lai contributes equally with Yifan Peng.

requests, leaving the queuing problem unaddressed.

In this paper, we explore a new design space: *parameter-centric memory management* based on two key observations. First, the massive computational requirements of model serving mandate the use of GPU clusters [45, 12, 35, 10], with model parameters replicated across multiple GPUs to handle workloads beyond a single GPU's computation capacity. Second, GPU memory is dominated by both KVCache and model parameters, with model parameters comparable (or even larger) than KVCache. Hence, when a GPU encounters memory throttling, pruning a portion of the parameters can free up sufficient HBM for queued requests. Such requests are then executed alongside ongoing requests with a larger batch size, eliminating the queuing delays. Since no KVCache is pruned, there is no tradeoff between current and incoming requests as in KV-centric approaches. The missing parameters can be borrowed from other replicas, such that requests can be processed with pipeline parallelism with incomplete parameters. The overhead of pipeline parallelism only requires an activation forward pass, which is negligible compared to the queuing.

However, parameter-centric memory management is non-trivial and faces several challenges. First, when parameters are dropped, many requests remain in processing due to the streaming execution nature of LLMs. Such requests cannot continue execution with dropped parameters since LLMs require a one-to-one mapping between parameters and KV-Cache for processing. While migrating the KVCache on demand can address this issue, such migration would stall request processing considering the huge amount of KVCache. Worse even, we cannot use state-of-the-art live KVCache migration techniques [45] because they require a complete copy of the model parameters on both source and target GPUs, which is missing due to our drop-based management. Second, pipeline execution comes at a cost: when more GPUs are cooperatively executing requests, the greater the overhead of pipeline parallelism. With an improper plan (how to drop parameters), even if we execute more requests concurrently with sufficient HBM, queuing can still occur because our GPUs cannot digest these requests timely. Thus, we need to come up with a plan that minimizes overhead while maintaining sufficient memory for queued requests.

To address the above challenges, we propose KUNSERVE, the first LLM serving system that uses parameter-centric memory management to cope with bursty memory throttling for LLM serving. To live migrate KVCache with incomplete parameters, we make a key observation: *in LLM computation, operators that require KVCache don't need parameters, and operators that require parameters don't need KVCache*. Therefore, during KVCache migration, we can offload the KVCache-related computation to the source GPU without interrupting current requests, which realizes live KVCache migration even with incomplete parameters. KUNSERVE further incorporates a greedy-based plan generation method to

quickly find a drop plan with minimal performance overhead. The observation is that the overhead positively correlates with the number of GPUs involved in the cooperative execution. Moreover, combining more GPUs in a cooperative group, though it can free more memory, yields diminishing returns. Thus, a greedy approach that minimizes instances in a group typically yields a reasonable plan quickly.

In addition to parameter dropping, KUNSERVE also dynamically restores dropped parameters when KVCache memory demand decreases, which is essential for maintaining high performance during normal load. Extensive experiments show that under various real-world traces and datasets, KUNSERVE achieves up to 13.6–27.3 × tail latency decrease and reduces 53.8–100% SLO violations compared to the state-of-the-art systems like vllm [29] and Llumnix [45]. In summary, this paper makes the following contributions:

- A new design space called parameter-centric memory management for coping with memory throttling in LLM serving (§3).
- A set of new techniques to make our parameter-centric memory management method efficient (§4).
- A set of evaluations confirming the effectiveness of the design (§5).

2 Background and Motivation

### 2.1 Preliminaries of LLM and LLM serving

LLM basics. LLM is a transformer-based [48] deep learning model. Compared with traditional DNN, a key difference is that it executes requests in an *auto-regressive* pattern with a *prefill* and *decode* phase. In the prefill phase, the input is fed to the model to generate the first token of the output. The decode phase then iteratively generates the rest of the output in a token-by-token way, where each iteration takes the previously generated token as well as the prefill input as the input. The decode[†] ends when the model generates a special end-of-sequence (EOS) token.

During decoding, since the same prefix of input is shared across all the iterations, the internal results (termed *KVCache*) are cached in the GPU memory (HBM) for acceleration. This makes the computation patterns of prefill and decode different [36, 27, 56]: the prefill is compute-bound, while the decode is memory-bound.

Because LLM is computationally intensive, LLMs are typically deployed on GPU servers, which process both the prefill and decode phases in a batched manner [53, 29] to improve the GPU utilization.

Serving metrics: TTFT and TBT. As the output tokens are generated iteratively, current systems serve requests in a streaming fashion, i.e., once a token is generated, it is immediately returned to the user. Thus, both the *prefill latency*

[†]We use the term *decode* to refer to the execution of a single iteration in the decode phase in this paper.
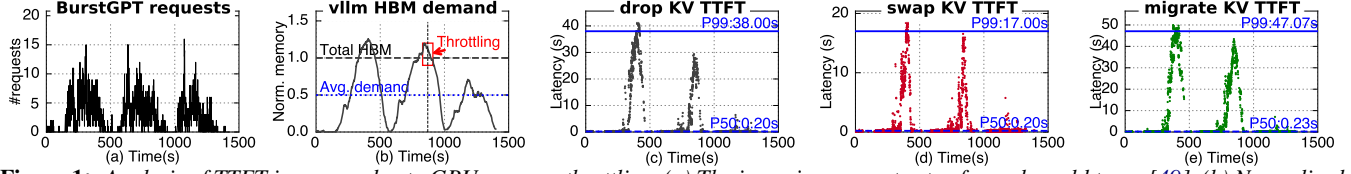
Figure 1: *Analysis of TTFT increases due to GPU memory throttling. (a) The incoming request rate of a real-world trace [49]. (b) Normalized KVCache memory usage on vllm [29] and (c)–(e) requests TTFT (normalized to input size) of existing solutions (§2.3).*
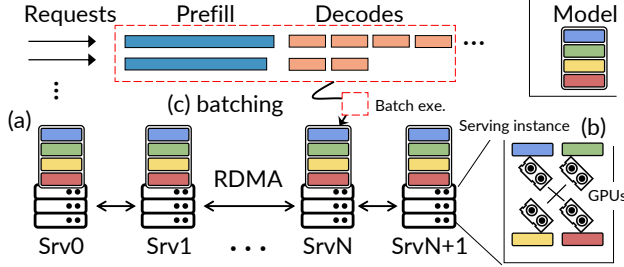
Figure 2: An illustration of a typical LLM serving scenario (a): how the model is deployed on each server with different serving instances (b) and (c) how prefill and decode requests are processed in a batched way. *exe.* is abbreviated for execution.

(**T**ime-**T**o-**F**irst-**T**oken, TTFT) and the *decode latency of each iteration* (**T**ime-**B**etween-**T**okens, TBT) matter.

Deploy LLM with parallelism and replication with serving instances. LLMs can be deployed on a single GPU or multiple GPUs with parallelism [30, 43, 55]. Pipeline parallelism (PP) partitions model parameters by layers, where layers belonging to the same group (called stage) are executed on the same GPU. Tensor parallelism (TP) partitions each layer, while different stages can reside on the same GPU. Parallelism comes at the cost of extra latency. For methods with high communication requirement like TP, parallelism is only applied to GPUs within the same server, because their interconnects are fast. PP on the other hand, can apply to GPUs across servers thanks to its ultra low communication volume. However, PP suffers from bubbles [6] especially for requests with a small batch size. TP and PP can be applied together.

In this paper, we term the minimal number of GPUs that have a single copy of the model parameters as a *serving instance*. The GPUs of an instance can be within the same server or across servers, but typically within the same server for the lowest serving latency unless the model is out of the capacity of a single server, which is rare (e.g., Llama-3.1-405B). There are typically multiple instances with replicated models, as shown in Figure 2, because a single instance has limited serving capacity.

2.2 Problem: TTFT spikes caused by memory throttling

Huge HBM demands and memory throttling of LLM serving. The overall memory demand for processing serving is huge. Considering serving a Qwen-2.5-14B model, where each token will consume 192 KB memory, an already small

amount due to GQA [7], a typical batch (e.g., 64 K tokens) on the BurstGPT trace will consume 12 GB HBM, not considering the retained memory of unfinished requests.

Since LLM serving is memory hungry, GPUs may meet memory throttling for two reasons. First, real-world traces exhibit spiked loads: Figure 1 (a) shows a real-world trace on BurstGPT [49], where the incoming request rate increases by $14\times$ at time 150s with no clear pattern, so the serving systems require a proportional KVCache demand to these requests. Even worse, for each request, its memory usage depends on how many tokens need to be generated. Since the tokens are generated auto-regressively by the model, its memory usage continuously grows and can throttle the GPU memory. For example, the average and variance of a BurstGPT [49] request stay time is 12 and 149 seconds, respectively[†].

Figure 1 (b) shows how existing serving system behaves under BurstGPT. During a 1,400-second period of serving, we observed two throttling happen on vllm [29]—a state-of-the-art LLM serving system, and the timing of throttling is strongly related to the request spikes. Note that we have chosen a practical setup where the overall HBM provisioned for KVCache is $2.1\times$ higher than the average requirement. We use a standard approach [45] that counts the memory demands by considering both the in-processing requests and head-of-line queuing requests.

TTFT spikes. GPU memory throttling is a killer for the serving performance. As shown in Figure 1 (c), the TTFT increases to up to $155\times$ after the throttling happens (see (b)). The increase comes from the queueing delays of waiting for sufficient memory to be freed up. The queuing time can be lengthy because the memory can only be freed once the ongoing request batch finishes. As we have mentioned before, the ongoing requests may take a long time to finish (up to 386s in BurstGPT).
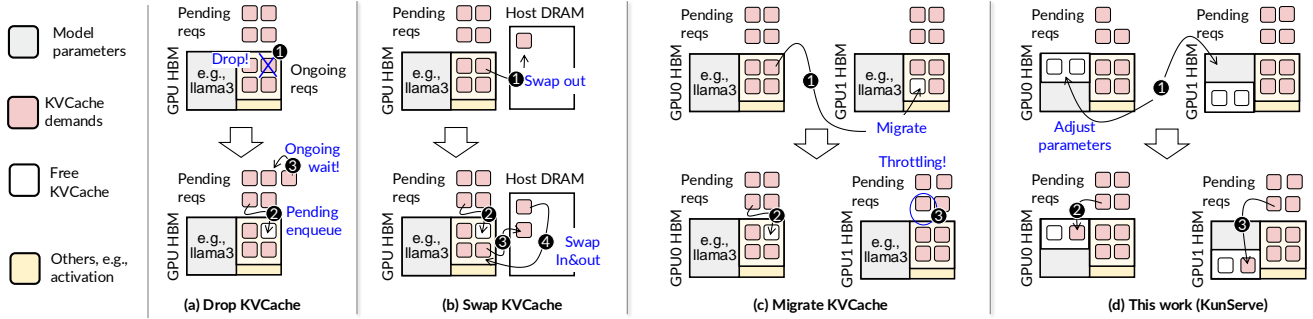
2.3 Existing KVCache-centric solutions

Figure 3 (a)–(c) shows an overview of existing solutions.

Drop the KVCache [29, 51, 39] (a). A naive solution is to drop some KVCache of ongoing requests (❶). Subsequently, queued requests can be processed with the freed GPU memory (❷). However, requests with dropped KVcache must be re-enqueued and recomputed, which also stalls incoming requests (❸) even without considering the recomputation overhead. As a result, Figure 3 (c) shows that simply dropping the

---
[†]Measured on an A100 GPU with Qwen-2.5-14B model.

Figure 3: Existing methodologies (a)—(c) to address memory throttling of KVCache and how KUNSERVE tackles this issue (d).
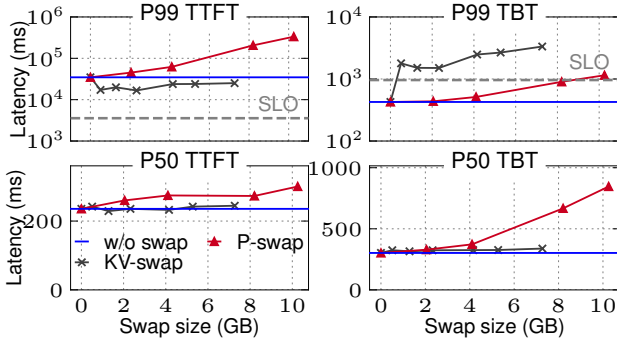
Figure 4: *A comparison of KVCache-centric swapping (KV-swap) and parameter-centric swapping (P-swap) on our motivation workload (BurstGPT) in Figure 1.*

KVCache still faces $190\times$ increases during memory throttling, even with a modest average memory load (49.8%).

Swap the KVCache [56, 29] (b). We can follow classic system solutions by swapping some KVCache to second-tier memory, e.g., host DRAM or SSD (❶), thus freeing up memory for pending requests (❷). However, frequent swapping in and out inevitably introduces overheads (❸). To control such overheads, existing systems only swap out a small number of requests (e.g., default 1 in vllm), otherwise it will introduce 3.5–$7.8\times$ P99 TBT increase, see Figure 4. This causes a dilemma: to avoid TBT increases, an optimal swap configuration in vllm still has a $85\times$ TTFT increase during memory throttling.

Migrate the KVCache [45] (c). Finally, observing that a serving cluster typically has multiple instances, a recent work (Llumnix [45]) migrates requests from a memory-throttled GPU to other (relative) spare GPUs (❶) for pending requests (❷). Note that migration is necessary to avoid fragmentation of the KVCache memory, so simply re-direct requests to spare GPU without migration is insufficient. However, until the migration is done, the queued requests can still be stalled. Worse even, under spike workloads, all instances' memory can be throttled (❸), opening up little room for migration. In Figure 3 (e), migration even exacerbates the issue, with P99 TTFT increased by $205\times$ (compared to the P50). This is because migration does not come free especially when it cannot help free up more memory.

Table 1: Popular LLM models and their GPU memory usage (on A100) deployed with a common setup.

| Model	Model size	#GPU per instance	Ratio (%)
Qwen-2.5-14B	28 GB	1 (80 GB)	34.4
Llama-3.1-70B	131 GB	4 (320 GB)	41.1
Qwen-2.5-72B	136 GB	4 (320 GB)	42.3
Llama-3.2-90B	180 GB	4 (320 GB)	56.3
Llama-3.1-405B	756 GB	16 (1,280 GB)	59.1

KVCache-centric vs. parameter-centric swap. While parameter swapping can also alleviate the issue, it is typically less considered because it impacts all the requests especially the TTFT. Figure 4 compares two swapping strategies. Note that we have adopted standard optimizations like asynchronous swapping and pipelined prefetching [21]. When swapping a few memory such as 0.5 GB, KVCache swap is always 1.1–$1.3\times$ faster than parameter swap for P50 TTFT, and up to $2.6\times$ faster for P50 TBT. This is because only 8.9–9.9% requests are impacted by KVCache swap while all requests are impacted by parameter swap. When swapping a larger amount of memory, the performance of both methods collapses, because they are all bottlenecked by the PCIe bandwidth.

3 Approach, Challenges, and System Overview

Our approach: parameter-centric memory management with cooperated execution. Unlike previous approaches that focus on adjusting KVCache memory, we found adjusting parameter memory—which consumes a significant portion (34–65%) of GPU memory (see Table 1)—can better handle memory spikes. This is because it allows us to instantly free up sufficient GPU memory to hold queued requests without affecting the KVCache of currently executing requests. Figure 3 (d) illustrates the idea: when memory throttling occurs and requests are queued on all the GPUs, we will drop half of the model's parameters on both GPUs (❶). This allows us to enqueue more requests on both GPUs (❷ and ❸) for execution with a larger batch size. Note that the average latency of requests increases due to large batch size, but the increases are much smaller than the queuing latency.

4
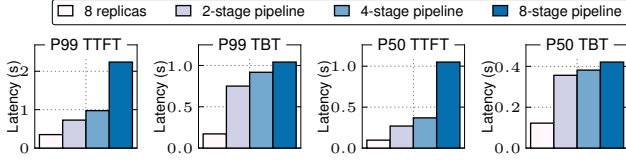
**Figure 5:** The comparison of replication and model parallelism on 8 GPUs running Qwen-2.5-14B model.
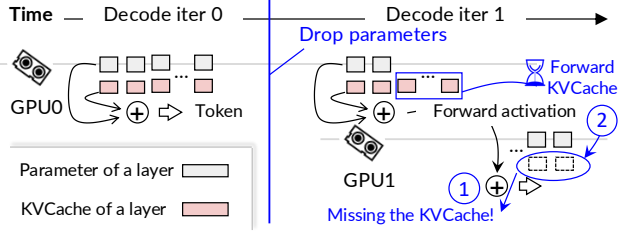
Figure 6: An illustration how an ongoing decode request becomes victim when we drop the parameters.

A problem of dropping parameter is that without a full copy of the model parameters, a GPU (or serving instance) can no longer complete request computations. We found this is not a problem in LLM serving systems: model parameters are typically replicated across multiple GPUs (instances) to handle workloads exceeding a single GPU/instance's capacity. As a result, we can cooperatively execute requests across multiple instances that have dropped parameters with pipeline parallelism. Figure 3 (d) exemplifies this: suppose initially the model is replicated on GPU0 and 1. If GPUs meet memory throttling, we can free the parameters of the second half of the layers on the GPU0, and free the first half on GPU1. Consequently, we can forward the queued requests to GPU1 for execution. After GPU1 finishes the first half of the computation, it forwards the activation to GPU0 for the remaining computation.

Why not always apply model parallelism. Readers might wonder why we don't always use pipeline parallelism (PP) for serving requests, as it can provide a larger KVCache capacity to better tolerate memory throttling. However, PP does not come for free: it introduces extra latency due to activation forwarding, and suffers from bubbles especially under normal workloads [5], leading to up to $1.8 \times$ higher P50 TTFT and $2.0 \times$ for P50 TBT (see Figure 5 and §5). Consequently, we only enable PP with parameter-centric memory management to handle memory throttling. We revert to fully replicated parameters for lower latency if the queuing is relieved.

Challenges. Though PP can enable instant processing of queued requests, it introduces a new challenge: how to handle ongoing requests (we termed *victim* requests) that have not been finished when we drop the parameters. These requests rely on the current cached KVCache to emit the next token. But since KVCache is associated with each layer, after dropping the parameters, we cannot simply compute the remaining layers on another GPU. Figure 6 illustrates this: In the first
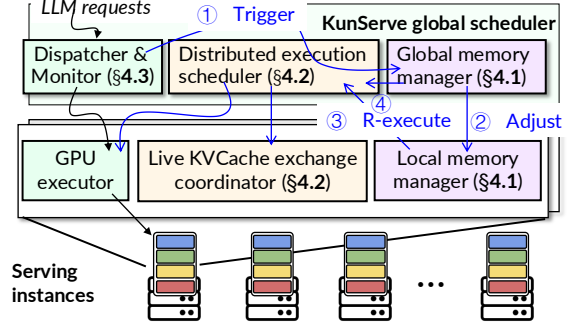
Figure 7: System overview of KUNSERVE.

decode iteration, a request executes on GPU0 has a full copy of model parameters. Before the start of the second iteration, we free up half of the parameters on GPU0, necessitating the forwarding of the remaining computation to GPU1. However, GPU1 lacks the last half of the KVCache (①), so it cannot execute the remaining computation of the request.

A naive solution is to kill the victim requests and recompute them. It is clearly inefficient due to GPU time wasted and re-enqueuing delays, especially as there may be many victim requests when GPUs are already overloaded. Another potential solution is to read the missing KVCache to the required GPU. For example, in Figure 6 (②), we could send the missing KVCache from GPU0 to GPU1 such that GPU1 can continue executing. However, it is still suboptimal because waiting for the KVCache transfer would significantly delay the request execution. The delay (1–8s on different network setups) is significant due to the large KVCache size. Existing live KVCache migration [45] cannot work in our case either because they require a full copy of the model parameters To address the issue, we propose live KVCache exchange, a new technique to enable efficient victim requests execution during dynamic parameter dropping by considering the execution nature of LLMs (§4.2).

Besides executing victim requests, PP does not come for free: while dropping more layers can free up more memory, the overhead also increases. For example, our evaluations in Figure 5 show that an 8-stage pipeline increases P99 TTFT by $5.4 \times$ and P99 TBT by $5.1 \times$ compared to no pipeline. Thus, another problem to address is how to generate an effective drop plan, with sufficient released memory to hold the queued requests, with minimal performance losses, while maintaining the invariant that at least one complete copy of the parameters exists across instances. Such a plan must also be generated in real time to handle sudden workload bursts (§4.1).

System architecture and overview. Figure 7 illustrates our system architecture. KUNSERVE is a cluster serving system that manages a set of LLM serving instances. The requests go through a global dispatcher that enqueues requests to the local executor of each instance for execution. The local executor inherits known techniques like continuous batching [53] and load balancing [45]. The global dispatcher monitors the memory usage of each instance and (§4.3), if necessary, triggers

the instances to drop (or restore) parameters (①) through a global parameter planner (§4.1). Once received the memory adjustment plan from the planner, each instance immediately drops the parameters accordingly for the KVCache (②). Note that in case of memory throttling has gone, KUNSERVE will also lively restore these parameters.

After the parameter has been dropped, queuing and ongoing requests need multiple instances' cooperation for execution, which necessitates global coordination (③). Hence, our distributed execution coordinator will work with the plan generated from the global parameter planner to (④) to effectively execute these requests. More importantly, it will work with our live KVCache exchanger (§4.2) on each instance, ensuring an overhead minimized transition when serving requests during parameter memory adjustment.

4 Detailed Design and Implementation

4.1 Parameter-centric memory management

Unlike traditional KVCache-centric memory management, where each instance manages its own GPU memory, our parameter-centric management uses a two-level global-local management strategy: we first generate a memory plan across instances, then execute the plan across involved instances locally. Such a two-level approach is necessary because, unlike KVCache, dropping parameters without coordination will cause LLM execution to fail.

Manage the memory globally with buddy groups. Upon receiving a notification from the workload monitor that one or more instances are likely to encounter (or have already encountered) memory throttling, the global manager decides how to drop the parameter across instances to free GPU memory for the KVCache. To ensure complete copies of the parameters always exist, we logically organize the instances into *buddy groups*, where each group is guaranteed to have exactly one copy of model parameters. Without memory throttling, each instance operates as its own buddy group. During memory throttling, two or more instances can merge into a single group, and requests in a group execute with pipelined parallelism.

With the group abstraction, generating a memory plan translates to finding a buddy configuration that has sufficient memory for queuing requests while minimizing overall latency increases. Specifically, the buddy configuration plan generation problem can be formulated as follows:

$$\text{Cost} = \frac{\text{Thpt. with group}}{\text{Thpt. without group}} = \frac{\text{Thpt}_I() \times N}{\sum_i^N \text{Thpt}_g(\sum_j^N G_{ij})}$$

where N is the number of instances (without drop) in the cluster, G_{ij} is 1 if the instance j is in the group i, and there would be at most N groups. $\text{Thpt}_I()$ is the throughput of a single instance with full parameter and $\text{Thpt}_g()$ is the throughput of a

group given a number of instances. For example, $\text{Thpt}_g(4)$ is the throughput of serving with a 4-stage pipeline parallelism. Both $\text{Thpt}_I()$ and $\text{Thpt}_g()$ can be profiled offline.

We denote the size of a replica of model parameters as P. The optimal group configuration needs to meet the following constraints:

(a) **Free memory** ensures the memory released by dropping the parameters is sufficient to hold the additional memory requirement (R).

(b) **Group surjectivity** ensures that every instance j is assigned to exactly one group i.

Put it all together, the optimization problem is listed below:

$$\text{minimize Cost}$$
$$\text{s.t. } R \le \sum_i ((\sum_j G_{ij} - 1) * P), \forall i, \sum_j G_{ij} > 0.$$
$$((a)\ \text{Free memory})$$
$$\sum_i G_{ij} = 1, j \in 1..N \qquad ((b)\ \text{Group surjectivity})$$

Finding the optimal configuration for the above program is hard due to non-linearity [46]: $\text{Thpt}_g()$ is a non-linear function, and the constraints (a) is also not linear. Fortunately, based on LLM serving features, we found a greedy method minimizing the number of instances in a group can achieve a near-optimal result. There are two greedy strategies.

Memory-greedy strategy. The strategy prioritizes the group configuration with more droppable parameter memory. To achieve this, we need to group as many instances as possible. However, our system cannot double the throughput by doubling KVCache capacity. As a result, we terminate the greedy process when the additional memory exceeds origin KVCache region.

Throughput-greedy strategy. The strategy prioritizes the group configuration with higher throughput. We achieve this by finding the group configuration that can just satisfy the memory demand of history requests (see Algorithm 1).

We choose the throughput-greedy strategy in our system. This is based on two observations. First, in LLM serving, the more instances in a group, the higher the inference cost due to more pipeline stages and bubbles [6, 56]. Moreover, when the number of instances in a group increases, there are diminishing returns to increasing the instance number in a group. For example, considering we have a cluster with 8 instances. If we divide them into 4 groups, then the freed memory is $4\times$ the parameter size. If we increase the instances per group by grouping all 8 instances into one group, the overall free memory only increases by $1.75\times$ ($7\times$ more memory).

Specifically, Algorithm 1 outlines our greedy algorithm group assignment process. We start from a state where each instance is in its own group, and iteratively merge groups with the smallest size until the memory constraint is met. The complexity of the algorithm is $O(N \log N)$, where N

Algorithm 1: Find Buddy Group Algorithm

Input: G_{ij} is 1 if the instance j is in the group i, 0 otherwise. bg_i is the size of group i. P is the parameter memory size. R is the additional memory required. BG is a priority queue that pops the group in the reverse group size. Initially, $G_{ij} = 1$ for all $i = j$, $bg_i = 1$ for all i and BG contains bg_i for all i.

Output: Buddy group configuration.

```
1  while |BG| > 1 and ∑_i((∑_j G_ij − 1) * P) < R do
2  │   bg_i ← BG.pop(), bg_i' ← BG.pop()
3  │   bg_i += bg_i'              ▷ merge group i' to i
4  │   for j ∈ 1..N do
5  │   │   G_ij ← 1 if G_i'j == 1  ▷ update assignments
6  │   │   G_i'j ← 0 if G_i'j == 1
7  │   end
8  │   BG.push(bg_i)
9  end
```


Figure 8: The pagedattention GPU kernel signature [2] (a) and (b) CUDA virtual memory management APIs [4].

is the number of initial instances in the cluster, so it can be efficiently solved. The algorithm can exit with requirements unmet, so we will scale a new instance for fallback (described in §4.3).

Unified local GPU memory management with CUDA virtual memory APIs. After receiving the plan from the manager, the local memory manager will immediately release the parameters' memory and delegate it to the KVCache. A key problem here is that existing LLM GPU kernels (e.g., PagedAttention [29]) cannot effectively reuse the memory freed by the parameter, because they assume a continuous virtual address space to store the available KVCache, as shown in Figure 8 (a). Unfortunately, the memory freed by the parameter may not be continuous with the memory allocated for the KVCache (e.g., `k_cache_addr`). One possible solution is to rewrite these kernels to adapt to the newly available memory (e.g., it can use two KVCache buffers). However, efficiently rewriting LLM kernels is non-trivial: simple rewrites lead to performance drops that require months of iterative development to optimize [37]. Meanwhile, new KVCache kernels continue to emerge (e.g., MQA/GQA kernels [41, 8], flash decoding kernels [15, 14, 17]).

To be compatible with existing and future kernels, we propose a unified GPU virtual memory management for both the parameter and the KVCache. Specifically, when using the parameter memory for the KVCache, we will preserve the start pointer of the original KVCache buffer, and only en-

large its capacity. Hence the original kernel can work without changes. Doing so requires dynamically mapping the physical memory freed by the parameters to the KVCache's virtual address, which is made possible thanks to the recent releases of CUDA virtual memory management APIs (Figure 8 (b)). For example, `cuMemCreate` allocates a piece of GPU physical memory and `cuMemMap` can map it to an arbitrary virtual address. With these APIs, we will first allocate sufficient physical memory for both KVCache and parameter. If some parameter is dropped, we will map its backed physical memory to the tail of the current KVCache buffer's virtual address to enlarge the KVCache capacity. The dynamic mapping overhead is in the microsecond level [37], which is negligible compared to LLM inference.

### 4.2 Cooperated execution under parameter drop/restore

This section focuses on how we design cooperative execution to ensure efficient processing of both new and ongoing requests when we drop parameters to alleviate memory throttling. At the end we briefly describe how we restore parameters when the memory is no longer throttled.

Serve new requests after the parameter drop. After the buddy group is formed, our distributed execution scheduler will first select a buddy group based on existing load balancing policies [45], and forward incoming requests to the instance with the first half of the layers of parameters. Afterward, these requests will be served by instances in the group cooperatively with pipeline parallelism.

Serving victim requests with KVCache exchange. Unlike serving new requests, serving requests that have entered the decode phase during the parameter drop is more challenging. As we have mentioned in the overview, these requests cannot use pipeline parallelism for execution because the involved GPU may lack portions of the KVCache needed for execution. The KVCache can be missing bidirectionally between GPUs in a buddy group since different GPUs drop different portions of the KVCache, so we have to *exchange* KVCache between them. For example, consider two GPUs that initially have complete copies of the model parameters. To handle memory throttling, we form them as a buddy group and GPU0 drops the parameters for the first half of layers while GPU1 drops the second half. Ongoing requests on GPU0 lack the second half of KVCache when executed cooperatively on GPU1, while ongoing requests on GPU1 face a similar situation. To ensure smooth execution of ongoing requests, GPUs within a buddy group must exchange KVCache of ongoing requests, i.e., GPU0 sends half of its current KVCache to GPU1, and GPU1 does the same in return.

During KVCache exchange, ongoing requests must wait for the exchange to complete before execution, which is costly. For example, when serving a Qwen-2.5-14B model on A100 GPU, when parameters are dropped, we need to exchange 33 GB KVCache between two GPUs, taking 1.3 s with a
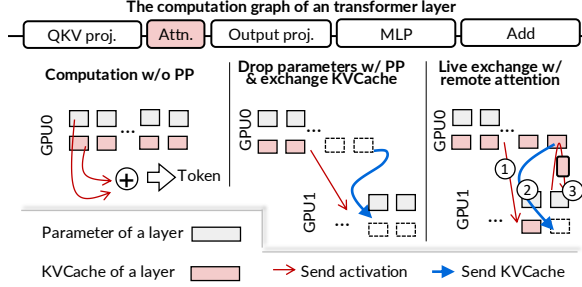
Figure 9: An illustration of the problem of live KVCache exchange and our remote attention-based solution.

200 Gbps GPU-direct RDMA network. An ideal execution should be *live* where ongoing requests can continue execution. Live exchange is similar to live KVCache migration, but due to the dropping of parameters, existing techniques become obsolete in our case. For example, Llumnix [45] proposes a pre-copy-like live KVCache migration technique that executes requests on the source GPU while migrating the KVCache to the target. It is not feasible for us because the source GPU lacks parameters to continue execution. Another solution is to borrow the idea of post-copy from virtual machine live migration [25]: we move the missing KVCache on-demand on the target GPU. However, this solution is not more efficient than a stop-of-the-world migration, because a token can be emitted only if all the KVCache is transferred.

Live KVCache exchange with remote attention. To realize live KVCache exchange, we propose remote attention that leverages the computing features of LLMs: not all computations in a transformer layer need KVCache, and computations that need KVCache do not need the parameters. The upper part of Figure 9 shows 5 major operators in a transformer layer: only the attention operator requires KVCache. Moreover, the attention is a fused scaled dot-product with softmax [48], which requires no parameter other than the KV-Cache[†]. As a result, if an instance misses the KVCache of the layer during exchange, we can schedule the attention to be executed on the source (remote) GPU without stopping the process of ongoing requests.

The right part of the Figure 9 shows a concrete example. Suppose GPU0 and GPU1 form a buddy group, and a request has originally been processed on GPU0 before the drop of parameters. When we drop the parameter on GPU0, we will schedule the computation of layers with dropped parameters on GPU1 for execution (①). The KVCache of the dropped layers will be exchanged to GPU1 concurrently with the request execution (②). During the execution, if it executes the attention operator, while the corresponding KVCache has not been transferred to GPU1, we will send the activation back to GPU0 for execution (③). Note that the attention result should be sent back to GPU1 for executing the next operator.

Remote attention and network requests coordinations. Im-

[†]The scaled dot-product operation does require a small scale parameter, but its memory footprint is negligible.

plementing remote attention efficiently is non-trivial: we need to carefully coordinate the network requests between GPUs to prevent interference. As shown in Figure 9, three types of network requests now must be transferred between GPUs: inter-layer activation for pipeline parallelism (①), attention activation for remote attention (②), and KVCache transfer for KVCache exchange (③). These requests share the same network link, and without proper coordination, the KVCache exchange will cause head-of-line blocking that prevents remote attention execution. For example, for a Qwen-2.5-14B model, exchanging the KVCache for 1 K tokens need to transfer a bulk of 192 MB data. In comparison, the activation of remote attention is only 672 KB.

To ensure live execution, we prioritize transfer activations over KVCache. Doing so requires transferring KVCache in a smaller granularity such that once the activation needs transferring, we can preempt the current KVCache transfer. Choosing the right granularity needs to take some care: it should be large to ensure a full bandwidth utilization, while it cannot be too large to avoid blocking KVCache. We leverage another fact of LLM serving for determining the right granularity: the time to transfer the KVCache, the activation and computation time of operators are relatively static and can be profiled statically. Based on these profiles, we calculate the maximum KVCache transfer granularity to be the one that can fit the interval between activation transfer, and stop transferring KVCache once an activation transfer is needed.

Live KVCache restore. When the memory demand decreases, we restore the dropped parameters on the involved instances, allowing them to serve requests locally without network communication. This results in lower latency. The parameters can be read from peer instances or from the host memory. However, due to the large size of parameters, the parameter restoration takes considerable time. This causes a problem: during restoration, requests must continue using pipeline execution, so when restoration completes, many requests are still processing through the pipeline, causing sub-optimal performance.

One solution is to redirect these requests to an instance with complete parameters for processing. However, the chosen instance can miss the KVCache of the ongoing requests, similar to the KVCache exchange problem. Thus, we also need to perform a live KVCache restore for the ongoing requests. Fortunately, handling live restore is simpler because all instances have the required parameters. Thus, we can either retrofit Llumnix [45]'s live KVCache migration or our proposed remote attention. Empirically, we choose a retrofitted version of Llumnix's live KVCache migration because it requires fewer network communications than remote attention. Specifically, once the KVCache is restored on the chosen instance, the ongoing requests can still execute with pipeline. Once the transfer is done, they will exit the pipeline.

4.3 Online dispatcher, monitor and others

Online dispatcher. Similar to existing cluster LLM serving systems [45], our dispatcher performs load balancing and prioritizes requests based on their priorities. Differently, our dispatcher also re-balances requests after parameter dropping to minimize queuing latency. A re-balance is necessary because, after parameter dropping, although an instance can serve all its queued requests within its memory, the insufficient computational power of this GPU may still cause queuing delays. To prevent such queuing, we set a maximum batch limit for each instance. This limit can be profiled offline thanks to the static performance features of LLM inferences. For requests that cannot be executed in the current maximum batch (overflowed), we will send them back to the dispatcher for a re-balance. Specifically, after executing the parameter drop plan from §4.1, the dispatcher collects information about each instance's current and overflowed load. It then redistributes the overflowed load across available spare instances.

Autoscaling. The parameter-centric approach has limitations in handling memory throttling, e.g., there is a finite amount of memory that can be freed by dropping parameters. Like prior work [45, 18], we can automatically start new instances on demand in such cases, though efficient model scaling is beyond the scope of our paper. But it's worth noting that KUNSERVE works seamlessly with model autoscaling—we can process more requests during model scaling, a time-consuming process compared to LLM inference [36].

Online monitor and drop trigger. Our monitor periodically checks instances' memory pressure (i.e., used KVCache vs. available HBM) and triggers the parameter drop if necessary. We employ a threshold-based trigger policy similar to those used in scaling serverless functions [19]. Specifically, we calculate whether future instances will meet memory throttling with the following two rates: the increase rate of KVCache requirements and the change rate of available HBM. When the future KVCache usage is about to exceed the available HBM in one second, we will trigger the drop. We use such a tight threshold (one second) that only triggers the drop under throttling or near-throttling for two reasons. (1) Our drop mechanism works instantly, so early drops provide few benefits, and (2) dropping too early may harm performance, especially with false positives, because pipeline execution does not come for free. We leave the exploration of more complex policies, such as time series forecasting-based ones [40], as future work.

Fine-grained KVCache block management. To enable elastic KVCache management, existing systems [29] allocate and deallocate KVCache in fixed-sized blocks, where the block size is a constant related to the model size. This causes internal fragmentation in KUNSERVE: because we may shrink the model size through parameter dropping dynamically, so the KVCache block will shrink accordingly. Thus, we implement a fine-grained GPU HBM allocator based on buddy memory allocation [28], where block granularity shrinks to layer granularity, the smallest unit of parameters on a serving instance. This eliminates the internal fragmentation.

Fault tolerance. Unlike traditional LLM serving where failures between instances are isolated, since the parameters are fully replicated, in KUNSERVE an instance failure can disrupt other instances if they are in the same group. To ensure serving capability under partial instance failures, we re-plan parameters globally once a failure is detected. By replicating parameters in host DRAM, we can always ensure successful parameter restoration after a failure.

5 Evaluation

5.1 Evaluation setup

We have implemented KUNSERVE from scratch with 11K C++. We chose C++ as our core GPU memory management and local scheduler implementation, thanks to its fine-grained control of CUDA memory and network transfer, and GPU kernel executions. Though Python is dominant in LLM serving systems, we found its I/O coordination is too slow (e.g., ms-level with async-io) and cannot fit our requirements. Our fine-grained control includes careful scheduling of remote attention requests and normal pipeline requests, piggybacked processing different requests together to utilize the GPU's idle time, and efficient live KVCache exchange overlapping. Note that we reuse (Python) modules from current serving systems like Llumnix [45] and vllm [29] for global dispatchers and efficient GPU kernels.

Models. We use state-of-the-art models including Qwen-2.5-14B and Llama-3.1-70B throughout our evaluations. For Qwen-2.5-14B, each serving instance originally uses 1 GPU while Llama-3.1-70B and Llama-3.2-90B use 4 GPUs per instance.

Testbed. For single-GPU workloads (i.e., Qwen-2.5-14B), we evaluate KUNSERVE on 8 servers each with one NVIDIA A800 80GB GPUs, 128 CPUs, and 2TB host memory. The servers are connected via 200 Gbps RDMA network. For evaluations on Llama-3.1-70B and Llama-3.2-90B, we use another cluster that have 2 nodes each with 8 NVIDIA A800 80GB GPUs. GPUs within one server has 400 GB/s NVLink bandwidth, while each GPU across servers have 100 Gbps RDMA network.

Metrics. Like priori works [56, 29, 52, 45]. we focus on the TTFT and TBT of requests for our evaluations. For TBT, we calculate TBT of a request as the average decode time of all its tokens generated in decode phase. For TTFT, we directly reports the measured time when popping out the first token. We also report the SLO violations of TTFT and TBT. Similar to previous works [56, 38, 44, 16], we set a tight TTFT and TBT SLO as $5\times$ of their P50 values of requests when the system is under a modest load. This is because our workloads
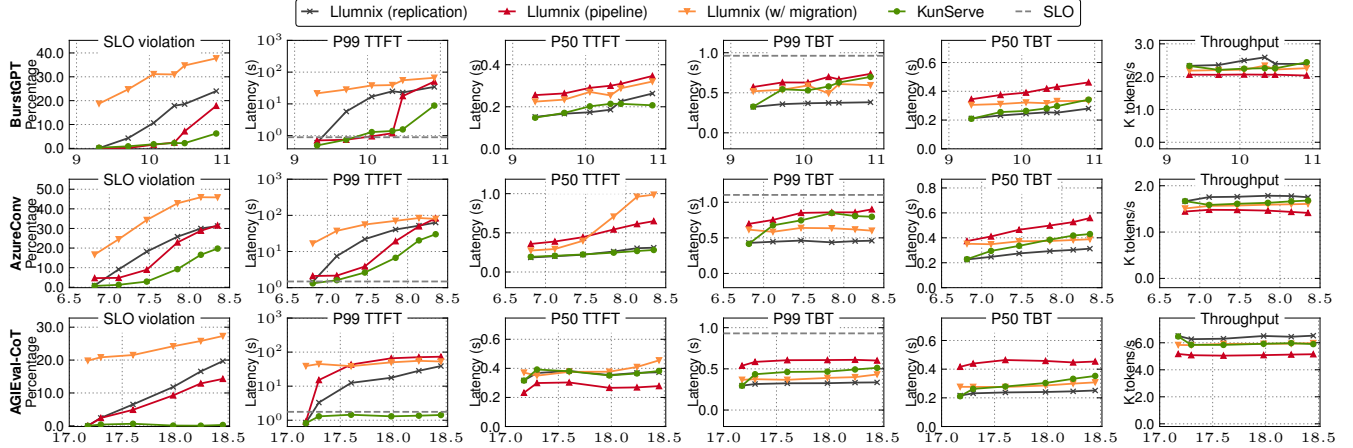
Figure 10: The end-to-end performance of KUNSERVE running Qwen-2.5-14B on different different datasets.

Table 2: Distributions of input and output lengths of various datasets from different serving scenarios.

Dataset		Mean	P50	P80	P95	P99
BurstGPT	In	603	400	1146	1804	2009
	Out	289	249	462	715	1161
AzureConv	In	1013	998	1159	4080	4096
	Out	247	202	411	451	585
AGIEval-CoT	In	237	128	556	619	651
	Out	390	363	450	578	698

Figure 11: The performance of KUNSERVE on BurstGPT trace.

all require low-latency responses (described below).

Evaluated traces and datasets. As the timing of throttling highly depends on the incoming requests pattern as well as how each request process, we choose real-world trace for the request arrival rate, and select different datasets to run on the selected traces. Specifically, we select the widely chosen BurstGPT trace [49]. As the request rate depends on the workload and cluster scale, we scale the incoming rate to fit our cluster following the instructions of their paper and priori work [49, 36, 33].

For datasets, we choose three representative datasets based on their patterns: decode-heavy workloads, prefill-heavy workloads, and balanced workloads. The prefill and decode lengthy of these workloads are summarized in Table 2, while the detailed workload type is described below:

- **BurstGPT.** It's the default dataset of the BurstGPT trace, which contains chatbot requests [49] with mean input length of 603 and mean output length of 289, placing it in the balanced workload category.

- **AzureConv.** It represents a real-world conversation application [36], characterized by a longer mean input length of 1013 and mean output length of 247, making it a prefill-heavy workload.

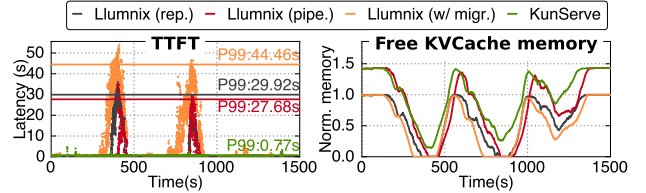- **AGIEval-CoT.** AGIEval-CoT is a QA dataset augmented

by chain-of-thought [47]. It has a mean input length of 237 and mean output length of 390. It is a decode-heavy workload.

Baselines. We mainly compare our system with Llumnix [45] (and its variants), a state-of-the-art cluster-scale LLM serving system. Like current serving systems [36, 56, 5], Llumnix uses a static parameter configuration that replicate parameters on all the instances by default. It uses global load balancing (**Llumnix (replication)**) and live KVCache migration (**Llumnix (w/ migration)**) to handle memory throttling. Finally, we also compare Llumnix with a static pipeline parallelism as **Llumnix (pipeline)**, where the pipeline stage number is set to the buddy group size found by our algorithm under one memory throttling 1.

5.2 End-to-end evaluation of KUNSERVE

Overall performance. Figure 10 shows the overall performance on 3 datasets with different request rates. To analyze the performance under various request rate, we choose a request rate with average memory requirements during the trace period and continuously increase the request rate. Overall, compared to Llumnix (replication), KUNSERVE reduces P99 TTFT by up to 8.4–27.3 × on different datasets We also achieve 2.9–52.2 × and 23.1–47.6 × better P99 TTFT compared to Llumnix (pipeline) and Llumnix (migration), respectively. These improvements reduce SLO violation from 32.9% to 0% on BurstGPT, 39.1% to 7.5% on AzureConv, and 27.8% to 0.6% on AGIEval-CoT at best cases, respectively.
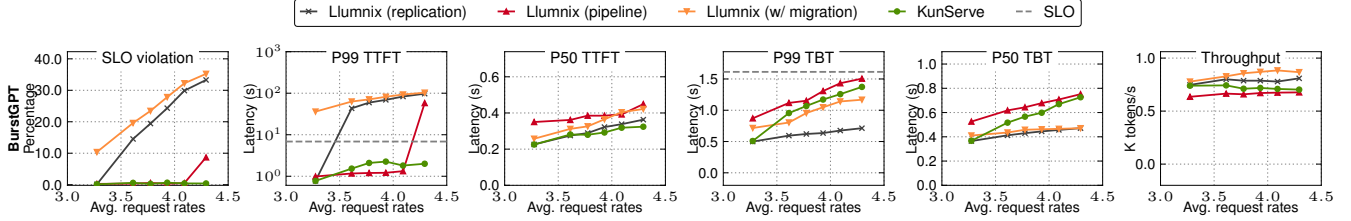
Compared to Llumnix (replication), the improvements

Figure 12: The end-to-end performance of KUNSERVE running Llama-3.1-70B on multi-GPU setups.
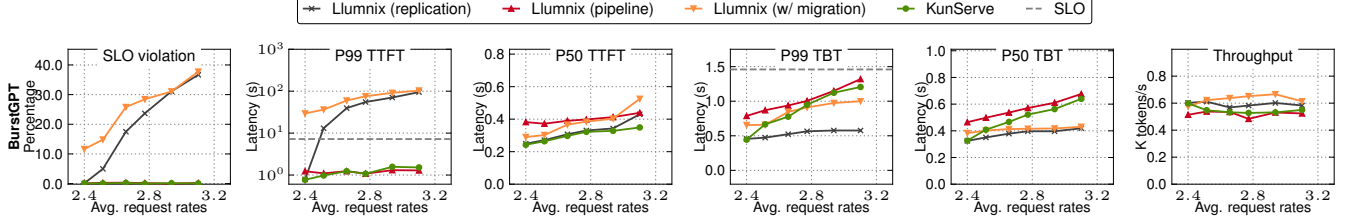

Figure 13: The end-to-end performance of KUNSERVE running Llama-3.2-90B on multi-GPU setups.

mainly come from reduced queuing under memory throttling. For example, when running BurstGPT dataset, we observed a 38.9 × TTFT increase when memory throttling happens as shown in Figure 11. Compared to Llumnix (pipeline), we are better in all metrics because pipeline introduces 1.7–1.8 × higher TTFT latency and 1.6–2.0 × TBT latency when the system is unloaded (12.6–21.8% lower in throughput). Interestingly, a pipeline configuration—though with more HBM for KVCache—can cause 35.9 × P99 TTFT increase. The only exception is AGIEval-CoT dataset, where we observed Llumnix (pipeline) has 1.4 × faster P50 TTFT compared to Llumnix (replication). This is because AGIEval-CoT has fewer input tokens and pipeline parallelism benefits from its multi-queue scheduling. However, KUNSERVE is still 1.2–2.0 × better in P50 TBT and 1.1–1.8 × better in P99 TBT in this case. For Llumnix (w/ migration), as we have extensively analyzed in §2.3, it can exacerbate memory throttling issue and thus has the worst performance.

Though KUNSERVE reduces the tail TTFT of all workloads significantly, it comes at a little cost of P50 and P99 TBT, with 7.9–37.5% and 33.4–77.4% increases on different datasets compared to the fastest baselines (Llumnix (replication)), respectively. These increases in latency arise because the requests that enter the system after parameters have been dropped suffer from pipeline overhead. The largest TBT increase is observed in AzureConv trace, where prefill phase is longer and thus the pipeline parallelism generates more execution bubbles. However, we believe such a trade-off is reasonable because no SLO is violated due to TBT increase even with a tight SLO ratio for TBT.

Multi-GPU performance. To show the generality of our approach, we also evaluate KUNSERVE with large models that require multiple GPUs. We run 4 Llama-3.1-70B instances on 2 nodes with 8 GPUs each. The results are shown in Figure 12, we report the results of BurstGPT dataset for brevity. Other datasets show similar trends. KUNSERVE can reduce
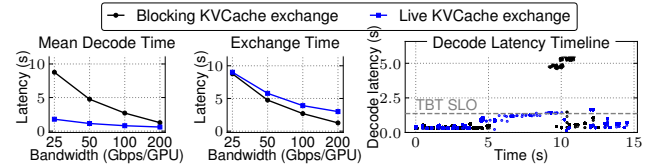

Figure 14: The effectiveness of live KVCache exchange on different network setups.

the tail TTFT by up to 48.7 × while introducing no overhead to P50 TTFT. For P50 TBT and P99 TBT, KUNSERVE introduces 25.4–54.1% and 60.8–92.6% overhead compared to Llumnix (replication) but better than Llumnix (pipeline). As Llama-3.1-70B takes a larger amount (41.1%) of HBM, Llumnix (pipeline) achieves the lowest TTFT as it has no KV-Cache exchange overhead. But it collapses due to its 16.4% lower throughput and KUNSERVE has 29 × faster P99 TTFT than it in same request rate. Our experiment on Llama-3.2-90B model has similar results, as presented in Figure 13. We achieve 13.4–51.4 × lower P99 TTFT compared to Llumnix (replication) and reduce SLO violation from 36.7% to 0.1% in the best case and achieve exactly the same performance when no request violates SLO.

5.3 Ablation Study

Effectiveness of live KVCache exchange. Live KVCache exchange is the key to preserve KUNSERVE token generation when system is exchanging KVCache among buddy instances. Live KVCache exchange is most effective when the system has a relatively weak inter-server network. For example, it can reduce token generation stall time in a 8-GPU server with one 200Gbps NIC (i.e., 25Gbps per GPU). We compare the performance of live KVCache exchange in different network setups with blocking KVCache exchange. In this experiment, we limited bandwidth between instances by limiting the visible NICs. The results are shown in Figure 14. We report mean decode time of ongoing requests as the main metric. It is calculated as the time passed since the last decode iteration
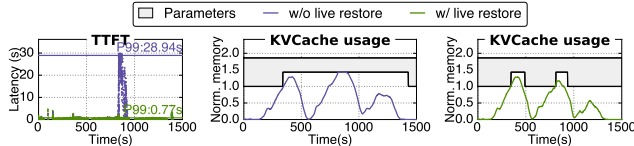
Figure 15: The performance of KUNSERVE with and without live KVCache restore.

Figure 16: The effectiveness of grouping strategies of KUNSERVE.

of the request. Like TBT, it represents decode performance. But it is different from TBT in that will not be amortized by other tokens of the request. We also present the token generation timeline under 50Gbps per GPU setup for a better understanding of the benefits. Blocking KVCache exchange has a generation stall up to 8.8s in the weakest setup, while live KVCache exchange has 51.7–79.8% lower mean decode time during KVCache exchange in all setups. TBT SLO violation means severe token generation performance degradation that affects user experience [11]. Notice that the decode time of tokens affected by KVCache exchange are far larger than TBT SLO during KVCache exchange. These tokens may not represent a significant portion of all tokens generated during LLM serving, but they are crucial because all users experience this second-level latency when the system is exchanging KVCache.

As a tradeoff, live KVCache exchange can prolong the total KVCache exchange time by up to 2.4 ×, and the increment ratio is larger in a faster network setup. However, we argue the tradeoff is reasonable because the KVCache exchange time is not a killing metric for user experience but the decode time is, especially when we actually waste near no time during KVCache exchange.

Effectiveness of live KVCache restore. To avoid performance degradation of pipeline serving, KUNSERVE also utilizes live KVCache restore mechanism to minimize pipeline serving time. To show the effectiveness of the technique, we evaluate the performance of KUNSERVE with and without live restore. We report the results of Qwen-2.5-14B model running with BurstGPT dataset. Other datasets show similar trends. We report the results in Figure 15. Pipeline serving is slower than normal instance serving, therefore long pipeline serving is risky in that it may not finish ongoing requests before the next memory spike. In our experiment, we observe that disabling live restore can lead to 37.6 × higher P99 TTFT. We also achieve 30.7% reduction in P50 TTFT, 34.3% reduction in P50 TBT, and 9.1% reduction in P99 TBT due to minimized pipeline serving time. SLO violation drops from 4.0% to zero accordingly. The latency increase happens because system encounters memory throttling during the second memory spike. Live KVCache restore has 25.7% free memory during the second memory spike while the other setup has used up its KVCache memory. As a result, no request suffers from memory throttling in live KVCache restore. Therefore, live restore is crucial for KUNSERVE to maintain low serving latency during contiguous memory spikes.
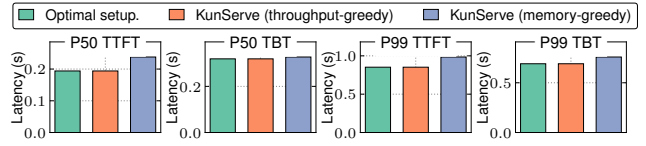
Effectiveness of buddy group algorithm. KUNSERVE uses a buddy group algorithm to decide how instances drop parameters together. To see how bad our system can be without a proper buddy group algorithm, we evaluate the performance of origin KUNSERVE that apply throughput greedy strategy to memory greedy strategy. We search the optimal performance by traversing all possible buddy group configurations and report the performance in Figure 16. We performance the evaluation on BurstGPT dataset with same setup in Figure 10. With throughput-greedy strategy, KUNSERVE successfully finds the best buddy group configuration and reach the optimal performance. Compared to the optimal setup, memory greedy strategy provides more KVCache memory but introduces more pipeline overhead. As a result, memory-greedy KUNSERVE has a 1.2 × higher P50 TTFT and 1.2 × higher P99 TTFT. For TBT, it has 1.1 × higher P99 TBT. Interestingly, the overhead is not caused by communication as we have seen little increase in P50 TBT, so we attribute the overhead to pipeline bubbles caused by more execution stages.

6   Related Work

Dynamic parameter reallocation. Spotserve also faces a similar parameter reallocation issue as we do, but we have a much tighter tolerance for reallocation overheads due to different scenarios. Specifically, Spotserve targets inference on spot instances, where reallocation is triggered passively when the platform revokes GPUs. Since the platform provides a relatively long grace period (e.g., 30s) for revocation, it can gracefully finish ongoing requests because the parameters aren't immediately changed. In KUNSERVE, our reallocation is online with tight requirements since the parameters are instantly dropped, and we propose live KVCache exchange approach to minimize token generation stall time.

Recent RLHF training works [42, 32] also reallocate parameters and reconfigure parallelism to improve training efficiency. However, their techniques is strongly coupled with RLHF workloads and cannot be directly applied to online serving. Their techniques are based on the observation that training process requires different parallelism configuration with generation process. Specifically, they use offline plan generator to find the optimal parallelism configuration for different phases of RLHF. We focus on online generation process, and our work is challenging because we have less tolerance for the overhead of parameter reallocation and parallelism plan generation. To achieve this, we propose a live KVCache exchange mechanism and a greedy online algorithm to minimize the search overhead.

LLM serving optimizations. Considerable research has focused on optimizing the computational processes and memory management of LLMs. From an algorithmic standpoint, FlashAttention [15, 14] leverages the GPU's memory hierarchy for I/O-aware attention mechanisms, achieving efficiency comparable to GEMM operations. From a scheduling perspective, SARATHI [6] and DeepSpeed-FastGen [26] refine prefill and decode phase scheduling, improving decode throughput. These optimizations are orthogonal to our work as they focus on improving computation during LLM serving while we focus on solving memory throttling. Another line of research solves the memory throttling problem by disaggregating prefill and decode operations, termed as PD-disaggregated systems. They are effective to meet strict SLOs [56, 36], but have strong assumption on inter-server network and have a lower resource utilization compared to our colocation setup.

PagedAttention [29] addresses KVCache fragmentation with fine-grained memory management APIs, achieving higher memory utilization. A recent work, vAttention [37] eliminates PagedAttention's overhead by using CUDA virtual memory APIs. We use the same APIs to extend KVCache memory. Lamina [13] offloads attention computations to cost-effective memory devices, with similar observation regarding the attention operation like us but it is orthogonal to our problem setting.

OS techniques for handling memory throttling. Handling memory throttling has been studied in operating systems. Linux utilizes the multi-generational LRU [1] mechanism for swapping pages between memory and disk. In disaggregated memory systems, researchers have suggested leveraging remote memory by offloading swap operations to kernel-bypass network devices [9, 24]. Swap sacrifices victim requests for better performance of new requests. It can alleviate TTFT increase but cannot solve the problem fundamentally. In this paper, we present a new parameter dropping approach with domain-specific observation of LLM serving to address this problem.

7 Conclusion

In this paper, we are the first to demonstrate that parameter-centric memory management can effectively address the latency spikes caused by memory throttling in LLM serving. We built KUNSERVE, an LLM serving system that cooperatively drops parameters to free up memory for processing queued requests while ensuring requests can still be correctly executed. This dramatically reduces the latency spikes caused by waiting for sufficient memory to be reclaimed for processing, because the dropped parameters can be instantly used by the queued requests with near-zero cost. We also propose techniques like live KVCache exchange, drop plan generation, and live restoration when we are building KUNSERVE. Our experiments show that KUNSERVE reduces tail TTFT by up to 27.3 ×.

References

[1] Multi-gen lru. https://docs.kernel.org/admin-guide/mm/multigen_lru.html, 2023.

[2] Easy, fast, and cheap llm serving for everyone. https://github.com/vllm-project/vllm, 2024.

[3] Introducing openai o1. https://openai.com/o1/, 2024.

[4] Virtual memory management. https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__VA.html, 2024.

[5] AGRAWAL, A., KEDIA, N., PANWAR, A., MOHAN, J., KWATRA, N., GULAVANI, B. S., TUMANOV, A., AND RAMJEE, R. Taming throughput-latency tradeoff in LLM inference with sarathi-serve. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024* (2024), A. Gavrilovska and D. B. Terry, Eds., USENIX Association, pp. 117–134.

[6] AGRAWAL, A., PANWAR, A., MOHAN, J., KWATRA, N., GULAVANI, B. S., AND RAMJEE, R. SARATHI: efficient LLM inference by piggybacking decodes with chunked prefills. *CoRR abs/2308.16369* (2023).

[7] AINSLIE, J., LEE-THORP, J., DE JONG, M., ZEMLYANSKIY, Y., LEBRÓN, F., AND SANGHAI, S. GQA: training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023* (2023), H. Bouamor, J. Pino, and K. Bali, Eds., Association for Computational Linguistics, pp. 4895–4901.

[8] AINSLIE, J., LEE-THORP, J., DE JONG, M., ZEMLYANSKIY, Y., LEBRÓN, F., AND SANGHAI, S. GQA: training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023* (2023), H. Bouamor, J. Pino, and K. Bali, Eds., Association for Computational Linguistics, pp. 4895–4901.

[9] AMARO, E., BRANNER-AUGMON, C., LUO, Z., OUSTERHOUT, A., AGUILERA, M. K., PANDA, A., RATNASAMY, S., AND SHENKER, S. Can far memory improve job throughput? In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020* (2020), A. Bilas, K. Magoutis, E. P. Markatos, D. Kostic, and M. I. Seltzer, Eds., ACM, pp. 14:1–14:16.

[10] ANYSCALE. Ray serve: Scalable and programmable serving. https://docs.ray.io/en/latest/serve/index.html, 2024.

[11] ARAPAKIS, I., BAI, X., AND CAMBAZOGLU, B. B. Impact of response latency on user behavior in web search. In *The 37th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '14, Gold Coast , QLD, Australia - July 06 - 11, 2014* (2014), S. Geva, A. Trotman, P. Bruza, C. L. A. Clarke, and K. Järvelin, Eds., ACM, pp. 103–112.

[12] AWS. Amazon bedrock. https://aws.amazon.com/en/bedrock/, 2024.

[13] CHEN, S., LIN, Y., ZHANG, M., AND WU, Y. Efficient and economic large language model inference with attention offloading. *CoRR abs/2405.01814* (2024).

[14] DAO, T. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations (ICLR)* (2024).

[15] DAO, T., FU, D. Y., ERMON, S., RUDRA, A., AND RÉ, C. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)* (2022).

[16] DELIMITROU, C., AND KOZYRAKIS, C. Amdahl's law for tail latency. *Commun. ACM 61*, 8 (2018), 65–72.

[17] ET. AL., T. D. Flash-decoding for long-context inference. https://pytorch.org/blog/flash-decoding, 2023.

[18] FU, Y., XUE, L., HUANG, Y., BRABETE, A., USTIUGOV, D., PATEL, Y., AND MAI, L. Serverlessllm: Low-latency serverless inference for large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024* (2024), A. Gavrilovska and D. B. Terry, Eds., USENIX Association, pp. 135–153.

[19] FUERST, A., AND SHARMA, P. Faascache: keeping serverless computing alive with greedy-dual caching. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021* (2021), T. Sherwood, E. D. Berger, and C. Kozyrakis, Eds., ACM, pp. 386–400.

[20] FURUTA, H., LEE, K., NACHUM, O., MATSUO, Y., FAUST, A., GU, S. S., AND GUR, I. Multimodal web navigation with instruction-finetuned foundation models. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024* (2024), OpenReview.net.

[21] GAO, B., HE, Z., SHARMA, P., KANG, Q., JEVDJIC, D., DENG, J., YANG, X., YU, Z., AND ZUO, P. Attention-store: Cost-effective attention reuse across multi-turn conversations in large language model serving. *CoRR abs/2403.19708* (2024).

[22] GIGASPACES. Amazon found every 100ms of latency cost them 1% in sales. https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales, 2024.

[23] GITHUB. Accelerate your development speed with copilot. https://copilot.github.com, 2024.

[24] GU, J., LEE, Y., ZHANG, Y., CHOWDHURY, M., AND SHIN, K. G. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017* (2017), A. Akella and J. Howell, Eds., USENIX Association, pp. 649–667.

[25] HINES, M. R., DESHPANDE, U., AND GOPALAN, K. Post-copy live migration of virtual machines. *ACM SIGOPS Oper. Syst. Rev. 43*, 3 (2009), 14–26.

[26] HOLMES, C., TANAKA, M., WYATT, M., AWAN, A. A., RASLEY, J., RAJBHANDARI, S., AMINABADI, R. Y., QIN, H., BAKHTIARI, A., KURILENKO, L., AND HE, Y. Deepspeed-fastgen: High-throughput text generation for llms via MII and deepspeed-inference. *CoRR abs/2401.08671* (2024).

[27] HU, C., HUANG, H., XU, L., CHEN, X., XU, J., CHEN, S., FENG, H., WANG, C., WANG, S., BAO, Y., SUN, N., AND SHAN, Y. Inference without interference: Disaggregate LLM inference for mixed downstream workloads. *CoRR abs/2401.11181* (2024).

[28] KNOWLTON, K. C. A fast storage allocator. *Commun. ACM 8*, 10 (1965), 623–624.

[29] KWON, W., LI, Z., ZHUANG, S., SHENG, Y., ZHENG, L., YU, C. H., GONZALEZ, J., ZHANG, H., AND STOICA, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023* (2023), J. Flinn, M. I. Seltzer, P. Druschel, A. Kaufmann, and J. Mace, Eds., ACM, pp. 611–626.

[30] LI, Z., ZHENG, L., ZHONG, Y., LIU, V., SHENG, Y., JIN, X., HUANG, Y., CHEN, Z., ZHANG, H., GONZALEZ, J. E., AND STOICA, I. Alpaserve: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023* (2023), R. Geambasu and E. Nightingale, Eds., USENIX Association, pp. 663–679.

[31] LIU, Z., LIU, H., ZHOU, D., AND MA, T. Chain of thought empowers transformers to solve inherently serial problems. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024* (2024), OpenReview.net.

[32] MEI, Z., FU, W., LI, K., WANG, G., ZHANG, H., AND WU, Y. Realhf: Optimized RLHF training for large language models through parameter reallocation. *CoRR abs/2406.14088* (2024).

[33] MIAO, X., SHI, C., DUAN, J., XI, X., LIN, D., CUI, B., AND JIA, Z. Spotserve: Serving generative large language models on preemptible instances. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024* (2024), R. Gupta, N. B. Abu-Ghazaleh, M. Musuvathi, and D. Tsafrir, Eds., ACM, pp. 1112–1127.

[34] OPENAI. Chatgpt. https://chatgpt.com, 2024.

[35] OPENAI. Openai api. https://openai.com/index/openai-api/, 2024.

[36] PATEL, P., CHOUKSE, E., ZHANG, C., SHAH, A., ÍÑIGO GOIRI, MALEKI, S., AND BIANCHINI, R. Splitwise: Efficient generative llm inference using phase splitting, 2024.

[37] PRABHU, R., NAYAK, A., MOHAN, J., RAMJEE, R., AND PANWAR, A. vattention: Dynamic memory management for serving llms without pagedattention. *CoRR abs/2405.04437* (2024).

[38] PREKAS, G., KOGIAS, M., AND BUGNION, E. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017* (2017), ACM, pp. 325–341.

[39] QIN, R., LI, Z., HE, W., ZHANG, M., WU, Y., ZHENG, W., AND XU, X. Mooncake: A kvcache-centric disaggregated architecture for LLM serving. *CoRR abs/2407.00079* (2024).

[40] SHAHRAD, M., FONSECA, R., GOIRI, I., CHAUDHRY, G. I., BATUM, P., COOKE, J., LAUREANO, E., TRESNESS, C., RUSSINOVICH, M., AND BIANCHINI, R. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *Proceedings of the 2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020* (2020), A. Gavrilovska and E. Zadok, Eds., USENIX Association, pp. 205–218.

[41] SHAZEER, N. Fast transformer decoding: One write-head is all you need. *CoRR abs/1911.02150* (2019).

[42] SHENG, G., ZHANG, C., YE, Z., WU, X., ZHANG, W., ZHANG, R., PENG, Y., LIN, H., AND WU, C. Hybridflow: A flexible and efficient RLHF framework. *CoRR abs/2409.19256* (2024).

[43] SHOEYBI, M., PATWARY, M., PURI, R., LEGRESLEY, P., CASPER, J., AND CATANZARO, B. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR abs/1909.08053* (2019).

[44] STOJKOVIC, J., ZHANG, C., GOIRI, Í., TORRELLAS, J., AND CHOUKSE, E. Dynamollm: Designing LLM inference clusters for performance and energy efficiency. *CoRR abs/2408.00741* (2024).

[45] SUN, B., HUANG, Z., ZHAO, H., XIAO, W., ZHANG, X., LI, Y., AND LIN, W. Llumnix: Dynamic scheduling for large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024* (2024), A. Gavrilovska and D. B. Terry, Eds., USENIX Association, pp. 173–191.

[46] SVEN LEYFFER, J. L. Mixed integer nonlinear programming (minlp). https://www.zccfe.uzh.ch/static/files/Munson_minlp.pdf, 2024.

[47] TAUR-LAB. Taur-lab/taur_cot_analysis_project___-gpt-4o-mini-2024-07-18. https://huggingface.co/datasets/TAUR-Lab/Taur_CoT_Analysis_Project___gpt-4o-mini-2024-07-18, 2024.

[48] VASWANI, A., SHAZEER, N., PARMAR, N., USZKOREIT, J., JONES, L., GOMEZ, A. N., KAISER, L., AND POLOSUKHIN, I. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA* (2017), I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, Eds., pp. 5998–6008.

[49] WANG, Y., CHEN, Y., LI, Z., KANG, X., TANG, Z., HE, X., GUO, R., WANG, X., WANG, Q., ZHOU, A. C., AND CHU, X. Burstgpt: A real-world workload dataset to optimize llm serving systems, 2024.

[50] WEI, J., WANG, X., SCHUURMANS, D., BOSMA, M., ICHTER, B., XIA, F., CHI, E. H., LE, Q. V., AND ZHOU, D. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022* (2022), S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds.

[51] WU, B., LIU, S., ZHONG, Y., SUN, P., LIU, X., AND JIN, X. Loongserve: Efficiently serving long-context large language models with elastic sequence parallelism. *CoRR abs/2404.09526* (2024).

[52] WU, B., LIU, S., ZHONG, Y., SUN, P., LIU, X., AND JIN, X. Loongserve: Efficiently serving long-context large language models with elastic sequence parallelism. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, SOSP 2024, Austin, TX, USA, November 4-6, 2024* (2024), E. Witchel, C. J. Rossbach, A. C. Arpaci-Dusseau, and K. Keeton, Eds., ACM, pp. 640–654.

[53] YU, G., JEONG, J. S., KIM, G., KIM, S., AND CHUN, B. Orca: A distributed serving system for transformer-based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022* (2022), M. K. Aguilera and H. Weatherspoon, Eds., USENIX Association, pp. 521–538.

[54] ZELIKMAN, E., HARIK, G., SHAO, Y., JAYASIRI, V., HABER, N., AND GOODMAN, N. D. Quiet-star: Language models can teach themselves to think before speaking. *CoRR abs/2403.09629* (2024).

[55] ZHENG, L., LI, Z., ZHANG, H., ZHUANG, Y., CHEN, Z., HUANG, Y., WANG, Y., XU, Y., ZHUO, D., XING, E. P., GONZALEZ, J. E., AND STOICA, I. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022* (2022), M. K. Aguilera and H. Weatherspoon, Eds., USENIX Association, pp. 559–578.

[56] ZHONG, Y., LIU, S., CHEN, J., HU, J., ZHU, Y., LIU, X., JIN, X., AND ZHANG, H. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024* (2024), A. Gavrilovska and D. B. Terry, Eds., USENIX Association, pp. 193–210.