KUNSERVE: Parameter-centric Memory Management for Efficient Memory Overloading Handling in LLM Serving

Rongxin Cheng, Yuxin Lai†, Xingda Wei✉, Rong Chen, Haibo Chen
Institute of Parallel and Distributed Systems, School of Computer Science, Shanghai Jiao Tong University

Abstract

Serving LLMs with a cluster of GPUs is common nowadays, where the serving system must meet strict latency SLOs required by applications. However, the stateful nature of LLM serving requires maintaining huge states (i.e., KVCache) in limited GPU memory. Under spikes in real-world workloads, GPU memory can be easily overloaded, leading to orders of magnitude higher response latency due to queuing introduced by waiting for KVCache to be reclaimed. Prior KVCache-centric approaches handle overloading by dropping, migrating, or swapping KVCache. These methods fail to release sufficient memory quickly with requests still queued.

This paper proposes the first parameter-centric approach to handling overloading by selectively dropping replicated parameters to instantly free memory for requests, based on an unnoticed observation that model parameters are commonly replicated across GPUs for serving LLMs. With additional memory, all requests can be served with a larger batch without queuing. To make the parameter-centric approach correct and efficient, we cooperatively execute requests on GPUs with a complete copy of parameters using pipeline parallelism, and derive an appropriate drop plan without unnecessary cooperation. We also design techniques to minimize the performance overhead due to pipeline parallelism with the execution patterns of requests under drop. Evaluations show that KUNSERVE reduces the tail TTFT of requests under overloading by up to 72.2× compared to the state-of-the-art systems including Llumnix, vLLM and InferCept.

CCS Concepts

• **Computer systems organization** → **Cloud computing**; • **Computing methodologies** → **Machine learning**.

Keywords

LLM Serving; Cloud computing; Parameter-centric memory management

EUROSYS '26, Edinburgh, Scotland Uk

ACM ISBN 979-8-4007-2212-7/2026/04
https://doi.org/10.1145/3767295.3769348

ACM Reference Format:
Rongxin Cheng, Yuxin Lai†, Xingda Wei✉, Rong Chen, Haibo Chen. 2026. KUNSERVE: Parameter-centric Memory Management for Efficient Memory Overloading Handling in LLM Serving. In *21st European Conference on Computer Systems (EUROSYS '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3767295.3769348

1 Introduction

Transformer-based large language models (LLMs) are reshaping the computing industry, which generate output in a token-by-token streaming fashion with auto-regressive inference. The tokens are used by downstream tasks such as chatbots [36], copilots [26], and interactive agents [24]. Such tasks require human interaction, so serving LLMs has tight latency requirements, e.g., less than 1 second [13, 55]. The smaller, the better [25]. Specifically, both the time to generate the first token (TTFT) and the time between subsequent tokens (TPOT) are important metrics.

A key feature of LLM inference is that the computation is *stateful*: before generating the final token, the intermediate results of previously generated tokens (termed *KVCache*) are kept in the scarce GPU memory (HBM) to accelerate future token generation. Such a stateful generation introduces a key issue: the serving latency could spike (up to 239× in BurstGPT [48], see §2.2 and others in §5) when the stored KVCache exhausts the precious HBM. Such overloading is common under real-world request bursts [23, 38] since the KVCache is proportional to the number of requests processed (or to be processed). Such overloading significantly impacts latency, because requests must wait for GPUs to free up sufficient memory for processing. Unfortunately, it could take seconds for LLMs to generate the final token so as to release memory due to the long and unpredictable token generation process.

State-of-the-art approaches adjust KVCache stored in GPU memory to handle overloading [30, 40, 44, 50]. When a GPU lacks sufficient HBM and causes request queuing, the system either drops KVCache of existing requests, swaps it out, or migrates it to an available spare GPU to make room for

†Work done while Yuxin was an intern at Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University. Yuxin was affiliated with Huazhong University of Science and Technology.

✉Xingda Wei is the corresponding author (wxdwfc@sjtu.edu.cn).

queued requests (detailed in §2.3). We argue that adjusting KVCache does not fundamentally resolve the queuing issue caused by memory overloading, because these methods do not release sufficient memory for all requests, i.e., they replace one set of queued requests with another. Thus, a portion of requests must still be queued, still resulting in sharp tail latency increases (e.g., more than 100×).

This paper answers a key question: *how can we effectively handle the latency spikes caused by memory overloading in LLM serving?* To answer this question, we propose a new system mechanism—parameter-centric memory management—to instantly free up abundant GPU memory upon overloading for all requests to eliminate queuing. Our method is motivated by two insights. First, the HBM usage is dominated by both KVCache and model parameters (34–74% per GPU, see Table 1), so dropping a portion of parameters can free up sufficient memory for processing all requests. While intuitive, dropping parameters inevitably disrupts the inference process, making the GPUs with dropped parameters unable to process requests. Thus, our second insight is that, due to the massive computational requirements of model serving, modern LLMs are served with a cluster of GPUs where the parameters are replicated across multiple GPUs [5, 6, 12, 14, 37, 38, 44]. As a result, as long as we carefully drop parameters to ensure complete copies exist cluster-wide, we can correctly process requests with dropped parameters using cooperative execution.

Our parameter-centric memory management operates in a three-step process. First, upon detecting that the serving system has suffered or is about to suffer from memory overload, we derive a drop plan and execute it across GPUs to free up sufficient memory. Afterward, requests executed on GPUs with dropped parameters are seamlessly rescheduled to groups of GPUs with complete parameters to ensure complete execution. These requests are executed using parallel inference techniques across GPUs with pipeline parallelism, since other techniques like tensor parallelism have more stringent network requirements. Finally, once the memory demand of the KVCache decreases, we restore parameters on the original GPUs and reschedule the requests accordingly to achieve the lowest inference latency.

Although the idea may appear simple, achieving parameter-centric memory management necessitates tackling a set of challenges. First, generating an efficient drop plan should holistically consider the memory freed up by the dropped parameters as well as the performance overhead introduced by dropping too many parameters. Meanwhile, we need a system mechanism to allow existing GPU kernels highly optimized for LLMs to use the HBM freed up by dropped parameters without modifications. To this end, we first leverage the predictable performance pattern of pipeline parallelism—the more parameters dropped, the more performance overhead incurred—to quickly derive a drop plan that minimizes the performance overhead while providing sufficient memory. Next, we design a unified GPU virtual memory management system with advanced GPU virtual memory features [4] to allow unmodified kernels to access the memory used for parameters for KVCache (§4.1).

Second, efficiently resuming requests after dropping requires exchanging KVCache between GPUs, since it is coupled with the parameters. However, such an exchange would significantly interfere with the pipeline-executed requests, because transferring large KVCache saturates the network used for forwarding activations. Observing that the activation transfer is more critical and the network usage is small, we design a coordinated network transfer engine that prioritizes the activation transfer to ensure both transfers are not affected (§4.2).

Finally, the pipelined execution across multiple GPUs after parameter dropping causes GPU bubbles [8], resulting in increased serving latencies and degraded throughput. The throughput degradation is particularly harmful in our setup, because if requests are processed at a slower rate, it could lead to another round of memory overloading. To tackle this problem, we identify the root cause of bubbles as suboptimal batch formulation in state-of-the-art systems like Sarathi-Serve [8]. By leveraging the observation that under overloading many requests are queued, we holistically form microbatches of queued requests using a new execution estimation metric combined with a lookahead batch formulation algorithm. Our scheduling minimizes the pipeline bubbles thanks to the holistic formulation during pipelined execution (§4.3).

We built KUNSERVE, the first LLM serving system with parameter-centric memory management. Under various real-world traces and datasets, when compared with the state-of-the-art baselines including Llumnix [44], vLLM [30] and InferCept [7], KUNSERVE achieves up to 12.7-72.2× tail latency reduction in these workloads, which further results in 7.2–12.8% lower SLO violations under common SLO factors. In summary, this paper makes the following contributions:

- A new parameter-centric memory management design for coping with memory overloading under LLM serving (§3).
- A set of new techniques to make parameter-centric memory management efficient (§4).
- Extensive evaluations confirming the benefits of KUNSERVE (§5).

KUNSERVE is open-sourced at https://github.com/SJTU-IPADS/kunserve.

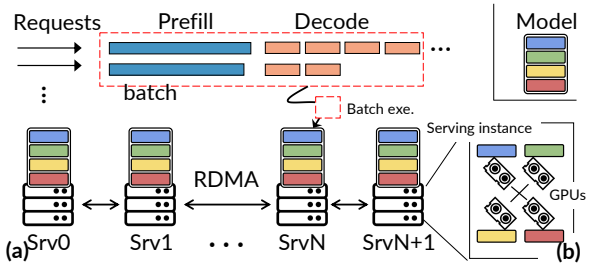

Figure 1: An illustration of a typical LLM serving scenario: (a) the model is deployed on different servers with model parallelism and prefill and decode requests are processed in a batched way. exe. *is abbreviation for execution.*

2 Background and Motivation

2.1 Preliminaries of LLM and LLM serving

LLM basics. LLM is a transformer-based [46] 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 context. The decode[1] ends when the model generates a special end-of-sequence (EOS) token.

During LLM inference, 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 [28, 38, 55]: the prefill is compute-bound, while the decode is memory-bound. To improve GPU utilization, modern LLM inference frameworks fuse prefill and decode requests into a single batch [8, 30].

Serving metrics: TTFT and TPOT. 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* (**T**ime-**T**o-**F**irst-**T**oken, TTFT) and the *time to emit each token* (**T**ime-**P**er-**O**utput-**T**oken, TPOT) matter.

Deploying LLM instances with parallelism and replication. LLMs can be deployed on a single GPU or multiple GPUs with parallelism [32, 42, 54]. Pipeline parallelism (PP) partitions model parameters by layers, where layers belonging to the same group (i.e., 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 requirements 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 [9] especially for requests with a small batch size. TP and PP can be applied together.

In this paper, we define the minimal set 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 exceeds capacity of a single server, which is rare (e.g., Llama-3-405B). For a serving cluster, a common practice is to deploy multiple instances with replicated models [5, 6, 38, 44], as shown in Figure 1, because a single instance has limited serving capacity.

2.2 TTFT Spikes from Memory Overloading

Huge HBM demands and memory overloading of LLM serving. The overall memory demand for LLM serving is huge. For example, when serving a Qwen-2.5-14B model, each token consumes 192 KB of memory, which is already relatively small due to the use of GQA [10], a memory-efficient attention mechanism. A typical burst still introduces an accumulation of 243 K tokens per GPU on BurstGPT trace (see Figure 2), consuming 45 GB KVCache memory per GPU.

We attribute GPU memory overloading to two causes. First, real-world traces exhibit spiked loads: Figure 2 (a) shows a real-world trace on BurstGPT [48], where the incoming request rate increases by 2 × at time 45s with no clear pattern. Since the KVCache demand is also proportional to the request rate, the memory demand can easily exceed the GPU memory capacity. Second, each request's KVCache may reside in GPU for a long time, with an unpredictable duration, depending on how long LLMs generate the EOS. For BurstGPT dataset, the average stay time for a request is 11 seconds, with a variance of 14.9 seconds. Thus, even the HBM is sufficient to hold incoming requests, GPUs still suffer from memory overloading due to the unfinished requests.

Figure 2 (b) shows how existing serving systems behave under BurstGPT. During a 640s serving period (§5.5), we observed two overloading events on vLLM [30], a state-of-the-art LLM serving system. The timing of overloading 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 × higher than the average requirement. We use a standard approach [44] that counts the memory demands by considering both the in-processing requests and head-of-line queuing requests.

TTFT spikes. GPU memory overloading severely degrades serving performance. As shown in Figure 2 (c), the TTFT

[1]We use the term *decode* to refer to the execution of a single iteration in the decode phase in this paper.

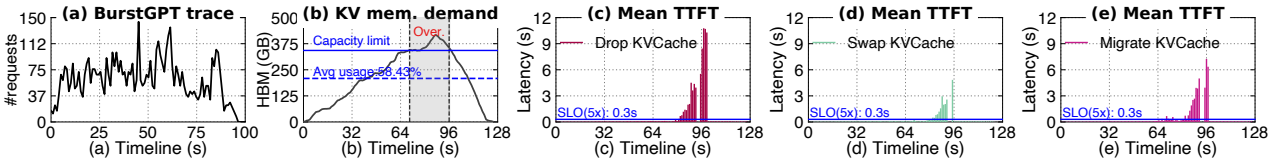

Figure 2: Analysis of TTFT increases due to GPU memory overloading (abbreviated as "Over." in figure). (a) The incoming request rate of BurstGPT trace [48]. (b) KVCache memory demand on vLLM [30] and (c)–(e) requests TTFT of existing solutions (§2.3).

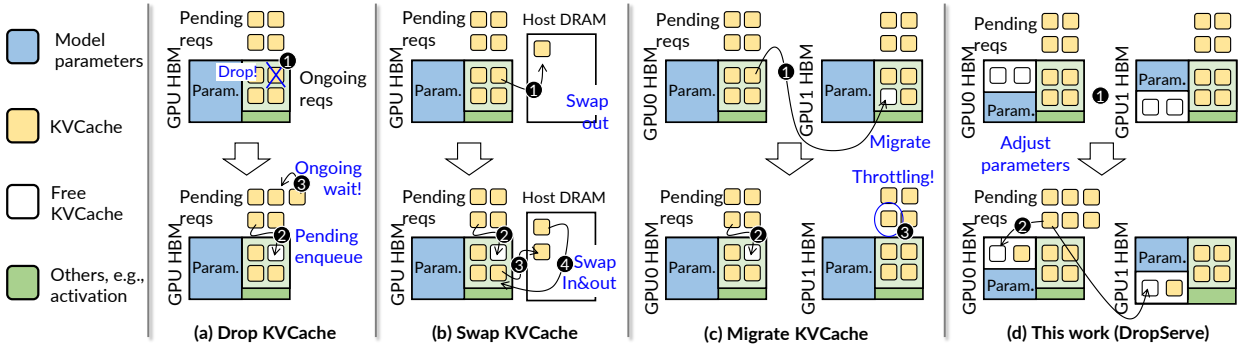

Figure 3: (a)—(c) Existing methodologies to address memory overloading of KVCache. (d) How KUNSERVE *tackles this issue via parameter dropping (❶) and remapping memory to enlarge KVCache region (❷).*

increases significantly after the overloading happens (see (b)). The increase comes from the queuing delays while waiting for sufficient memory to be freed. 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 (e.g., up to 150s in BurstGPT).

2.3 Shortcomings of Current Solutions

Drop the KVCache [30, 40, 50] (Figure 3 (a)). A straightforward 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 suffers the queuing overhead (❸) even without considering the recomputation cost. As a result, Figure 2 (c) shows that simply dropping the KVCache faces up to 239× TTFT increases during memory overloading, even with a modest average memory load (56.3%).

Swap the KVCache [7, 30, 52, 55] (Figure 3 (b)). A classic solution to handle memory overloading is swapping: when it happens, the system swaps out the overflowed KVCache to other storage (e.g., CPU DRAM) to free the GPU memory for execution (❶). The key problem is that as the GPU memory is still insufficient, there will inevitably be queued requests, even without considering the swapping overhead. For example, under overloading, InferCept [7] concurrently swaps out the KVCache of ongoing requests to hide the transfer overhead, but the queued requests are still waiting for ongoing requests to finish. The waiting time can be substantial because the overall decode time is orders of magnitude higher than TTFT. As a result, we still observed a 92× TTFT spike on InferCept [7] in Figure 2 (d). Worse still, the swapped-out requests (❸) further suffer high TPOT (see Figure 13).

Table 1: Popular LLM models, their parameter memory usage, the number of GPUs belonging to an instance, and the parameter memory usage ratio. Note that within an instance, Qwen-3-235B and DeepSeek-V3-671B are configured with expert parallelism with degrees 8 and 32, respectively, a common serving setup [20].

Model	Model size	#GPU/instance	Ratio (%)
Qwen-2.5-14B	28 GB	1 (80 GB)	34.4
Qwen-2.5-72B	136 GB	4 (320 GB)	42.3
Llama-3.1-405B	756 GB	16 (1,280 GB)	59.1
Qwen-3-235B	479 GB	8 (640 GB)	74.8
DeepSeek-V3-671B	1,572 GB	32 (2,560 GB)	61.4

Migrate the KVCache [44] (Figure 3 (c)). Finally, observing that a serving cluster typically has multiple instances, a recent work (Llumnix [44]) migrates requests from a memory-overloaded GPU to other (relatively) spare GPUs (❶) for pending requests (❷). The observation is that while no single GPU can hold all the pending requests, we can migrate requests to reduce fragmentation to free up sufficient memory. However, the queued requests can still be stalled because memory is occupied by migrating requests or the destination node is also memory-overloaded (❸). Worse still, under

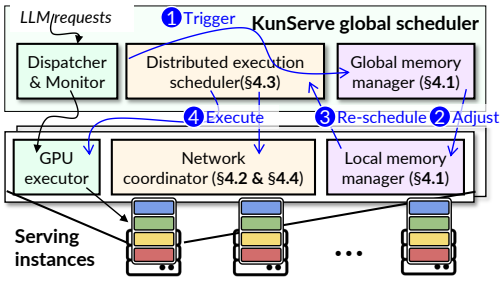

Figure 4: System overview of KUNSERVE.

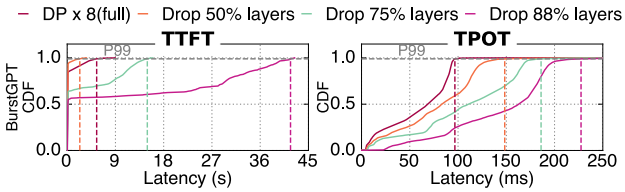

Figure 5: A comparison of the latency of different parallelism on BurstGPT dataset. All setups are evaluated with 8 GPUs.

spike workloads, there is little room for using migration to free up memory because the overall memory KVCache is insufficient even without considering fragmentation. Thus, as shown in Figure 2 (e), migration still leads to a 148× P99 TTFT increase (compared to the P50).

3 System Overview

Approach: online parameter dropping. As mentioned in the introduction, KUNSERVE is based on two key observations of LLM serving: (1) parameters typically take up a considerable portion of HBM per GPU (see Table 1) that can be used for KVCache and (2) parameters are replicated across instances so dropping them for KVCache does not impact LLM serving. Figure 3 (d) illustrates KUNSERVE's main approach and a comparison with other baselines assuming two instances and each instance uses one GPU. When the HBM used for KVCache is exhausted on GPU0 and GPU1, we instantly drop the second half of layers on GPU0 and the first half of layers on GPU1 (❶). Then, the queued requests are rescheduled on both GPUs (❷) for execution via pipeline parallelism.

Discussion: why pipeline parallelism? We chose pipeline parallelism because the network requirement can be easily satisfied with the interconnects between instances. Specifically, it requires orders of magnitude smaller communications than other parallelism setups that support execution after the parameter drop like tensor parallelism. While instances could link together via fast interconnects like NVLink for tensor parallelism, the domain of NVLink is much smaller than networks that could serve pipeline parallelism well like RDMA [35]. Thus, under overloading, we may be unable to find sufficient instances connected by NVLink.

System architecture. Figure 4 illustrates our system architecture as well as the workflow of parameter-centric memory management for handling memory overloading. KUNSERVE is a cluster-serving system that manages a set of LLM serving instances. Requests are routed through a global dispatcher, which enqueues them to the local executor of each instance for execution. Our dispatcher incorporates the load-balancing design from Llumnix [44]. The global monitor collects usage information and calculates the load metric for each instance.

Once a memory overloading event is detected by the monitor, it invokes our global memory manager (①) to generate dropping plans. The plan is then forwarded to the local manager on the involved instances (②) to adjust the memory according to the plan (details in §4.1).

After parameter dropping, KUNSERVE re-scheduled queued requests and ongoing requests to execute on instances with enlarged memory using pipelined parallelism (③). To ensure a smooth resumption of the requests whose KVCache is not on the target instances to avoid computation waste, our network coordinator exchanges the KVCache of ongoing requests between instances without blocking the activation transfer of pipelined execution (§4.2). Meanwhile, our optimized pipelined scheduling minimizes the bubbles in the upcoming execution (§4.3).

Finally, once the memory demand goes down, KUNSERVE dynamically restores parameters such that future requests can execute with lower latency. (§4.4).

4 Detailed Design and Implementation

4.1 Parameter Drop under Memory Overload

Upon overloading, KUNSERVE needs to generate a drop plan to free up sufficient memory. Besides the memory requirement, the plan has to meet the following requirements: (1) we need to generate the plan quickly online, (2) the plan needs to ensure a correct execution and (3) the plan needs to minimize the performance loss caused by parameters drop.

For (2), we only need to ensure that all the instances combined have a complete copy of parameters. However, dropping too many parameters incurs a performance cost. For example, suppose we are serving a 7-layer model with 7 instances. While dropping 6 layers on all instances can free 85 % of the HBM for KVCache, it forces the scheduler to split the batch into microbatches with smaller sizes, reducing the GPU batch execution efficiency [21] and making the system more vulnerable to pipeline bubbles. Figure 5 compares the serving latencies for different degrees of parameter dropping. We can clearly see that the more parameters are dropped, the higher the execution latency.

Input: $\boldsymbol{G} = \{g_0, g_1, ...\}$, existing group assignment,
$\boldsymbol{g_i} = \{I_0, I_1, ...\}$, instances belonging to a group,
$\boldsymbol{I_i} = \{l_0, l_1, ...\}$, layers belonging to an instance,
$\boldsymbol{R}$: the total memory requirement to free.

Output: a new group assignment.

```
1  freed = 0
2  Q = PriorityQueue(G, sortBy = |g|) ▶ min-heap
3  while |Q| ≥ 2 and freed < R:
4  |  g0, g1 = Q.pop_front(), Q.pop_front()
5  |  L_g0 = { l | l ∈ I, I ∈ g0 }
6  |  L_g1 = { l | l ∈ I, I ∈ g1 }
7  |  duplicated_layers = L_g0 ∩ L_g1
8  |  new_g = merge(g0, g1) ▶ Form a new group
9  |  freed += size(duplicated_layers)
10 |  Q.push(new_g)
11 return Q.to_set()
```

Figure 6: The pseudocode of drop plan generation algorithm.

A key takeaway from Figure 5 is that the performance loss is strongly correlated with the number of instances involved in processing a request, i.e., pipeline stages. Thus, we design a greedy-based parameter dropping algorithm by grouping as few instances as possible to minimize performance loss.

Algorithm 6 shows the details of our method that groups instances into groups to free up memory. The initial configurations (G) follow the setups without a drop, e.g., each instance itself is a group. To support greedy grouping, the group records the number of instances involved (g_i) and all instances are stored in a priority queue (Q).

Upon overloading, we first compute the memory demand of all queued requests (R) and enter line 1. Afterward, we iteratively group instances and then drop parameters to free more space (lines 3–9). For example, if there are three groups with sizes of 1, 2, and 3, we will select the two groups with sizes of 1 and 2 to form a new group (lines 5–6). For the selected groups, we drop a copy of the redundant parameters (line 7) and update the available memory (line 9). At the end of the iteration, the selected two groups are merged into a new group and inserted back into the priority queue (line 8).

The iteration continues until the memory requirement is satisfied or it fails to find a drop plan (line 3). In case we cannot find a plan, we fallback to the KVCache-centric solution to ensure continuous execution and autoscale the instance numbers. The complexity of the plan generation is $O(N \log N)$, so we can quickly execute it online even with a large number of instances.

Local instance memory management. A key challenge of executing the drop plan at each instance is how to allow existing attention kernels to use the freed parameter memory. As shown in Figure 7 (a), the kernels are written with a single

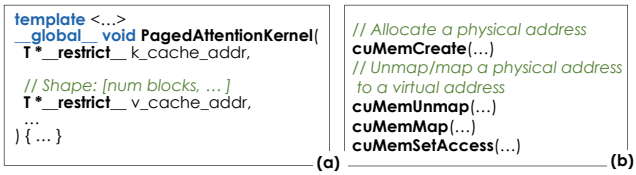

Figure 7: (a) The GPU kernel signature of the pagedattention kernel [2]. (b) CUDA virtual memory management APIs [4].

static memory layout, e.g., `[kcache_addr, kcache_addr + num_blocks * block_size]`, not multiple virtual memory ranges provisioned dynamically. One possible solution is to rewrite these kernels to suit the new memory layout. However, efficiently rewriting LLM kernels is non-trivial due to the complex and evolving nature of LLM kernels. Simple rewrites lead to performance drops that require months of iterative development to optimize [39].

To tackle the problem, we observe that recent GPUs have introduced application-controlled virtual memory management APIs: as shown in Figure 7 (b). For example, `cuMemCreate` allows allocating a piece of GPU physical memory and `cuMemMap` can map it to an arbitrary virtual address. With such APIs, we can dynamically change the virtual address space of KVCache without modifying the kernel code. The overhead of calling these APIs is in the microsecond level (5 ms on our platform), which is negligible to the LLM inference time. Specifically, our local instance memory management holistically manages the GPU physical memory for both the parameters and the KVCache with `cuMemCreate`. Afterward, when executing the drop plan received from the global coordinator, we first identify the physical memory of the dropped parameters. Then we extend the memory for KVCache by mapping the tail of the KVCache memory to the freed physical memory with `cuMemCreate`.

4.2 Smooth KVCache Transition of Requests

Because the KVCache has a one-to-one mapping with the model parameters, we cannot simply execute ongoing decode requests due to the lack of KVCache. For example, suppose a request has executed on instance A, and A has formed a group with instance B due to memory overloading. After the drop, A will only have parameters of layers 0–4, while B will have layers 5–7. Hence, B cannot directly execute the 5–7 layers of a request originally on A because the required KVCache is on A. Similarly, A cannot execute the 0–4 layers of a request originally on B. One intuitive solution is to recompute the KVCache on B. This is expensive since it causes queued requests to wait for the recomputation even without considering the recomputation time.

Network-based KVCache exchange. We choose to exchange the KVCache through the network to avoid recomputation. The KVCache is exchanged because after A and B

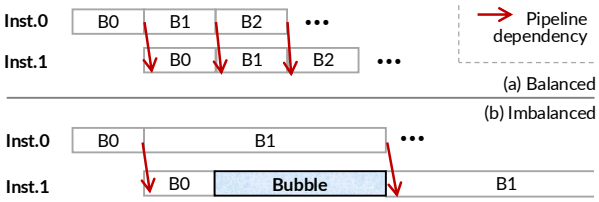

Figure 8: An illustration of pipeline execution bubbles caused by imbalanced execution time of microbatches.

have formed a group, ongoing requests on A need to transfer their KVCache to B, while B needs to do the same vice versa. A drawback of the exchange is that the requests with the exchanged KVCache will be stalled during the exchange, which we found to be acceptable in practice. This is because the network between instances such as RDMA is sufficient for transferring the KVCache quickly. For example, KVCache exchange typically introduces 1–2 s stall time on our 200 Gbps network. This means a 10 ms increase at most in the TPOT metric of a response with 200 decode tokens.

Note that during the stall, we can still schedule new requests queued due to memory overloading to fully utilize the GPUs. While in principle we can leverage techniques like attention offloading (also called model-attention disaggregation) [16] to concurrently execute stalled requests during the KVCache exchange, we found the excessive complexity of the implementation is not worth the effort.

Coordinated KVCache exchange. Although straightforward, KVCache exchange could block new request if not implemented properly, because the exchange competes for bandwidth with activation transfers in pipelined execution. Since the exchange time is much longer than forwarding the activation, When the activation is waiting for the exchange to finish, it will leave the GPUs idle, causing non-negligible performance loss. Observing that the activation transfer is much smaller yet more critical, we design a coordinated exchange mechanism to prioritize the activation transfer. Specifically, we transfer KVCache in finer-grained chunks such that the transferring a chunk takes similar time to executing a pipeline stage. After transferring one chunk, we will check whether there will be activation transfer. If so, we pause the KVCache transfer and let the activation transfer go first.

4.3 Efficient Serving after Parameter Drop

Key problem: pipeline bubbles caused by unbalanced microbatch execution time. A problem of pipeline execution after parameter drop is that the system suffers from degraded throughput due to pipeline bubbles. The bubbles arise from the imbalanced execution time of different microbatches, as illustrated in Figure 8 (b). For example, when B1's execution

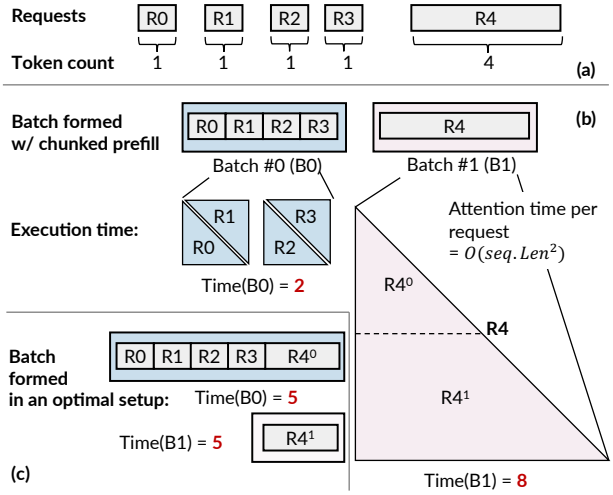

Figure 9: (a) An illustration of serving requests to execute. (b) The imbalanced batch execution time of existing chunking method. (c) A balanced formulated batch configuration.

time is longer than B0, Inst.1 must wait for B1 to finish before it can execute the layers on B2.

A preliminary on the state-of-the-art pipeline batching. Modern pipeline implementations rely on chunked prefill to reduce pipeline bubbles. Specifically, they [8, 30] form microbatches in a token-count-based manner, which balances the execution time of different microbatches by ensuring each microbatch has a similar number of tokens. As shown in Figure 9 (a), suppose 5 requests (R0–R4) arrive at an instance in turn, and the budget for each microbatch is 4 tokens. The scheduler first merges incoming requests into one microbatch (R0–R3 in (b)). R4 itself forms another microbatch (B1). Note that if R4 exceeds the budget, the scheduler will chunk it into two segments for execution.

Inefficiency of token-count-based chunking. A key issue is that the microbatch execution time is not linearly proportional to the total token count, because the attention computation of each request is quadratic to its token count, as shown in Figure 9 (b). Moreover, if a request is chunked into two parts, the latter chunk is slower than the former even when the tokens are the same, because the latter chunk has to additionally compute the attention with the former chunk.

The lookahead batch formulation. Fortunately, under bursts, we have sufficient requests queued. Thus, we can re-form the microbatches across them by looking ahead at all requests queued. To efficiently find the balanced microbatch configuration, we propose a heuristic divide-and-conquer algorithm.

Our method works in two steps. First, we adopted a retrofitted cost model to precisely estimate the execution time of a

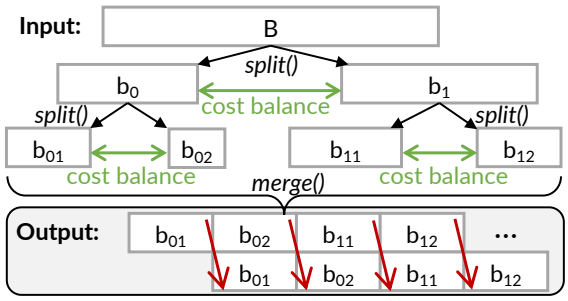

Figure 10: An illustration of how lookahead batch formulation recursively generate balanced microbatches.

Input: **B** = [[r0 || r || ...]], the initial batch contain one that has all requests ,
MIN , the minimal tokens per batch.
derived by dividing total token numbers, profiled off-line.
Output: a balanced micro batch set [b0, b1, ...,].

```
1  B = balance_micro_batch(B)
2  return B

3  Function balance_micro_batch(B):
4    if |B[0]| ≤ MIN:
5    |  return B ▶ Don't chunk if with few tokens
6    res = []
7    For b in B:
8    |  b0, b1 = b.split(0.5 * cost(b))
9    |  res = res || balance_micro_batch(b0)
10   |  res = res || balance_micro_batch(b1)
9    return res
```

Figure 11: The pseudocode of the divide-and-conquer microbatch formulation algorithm.

microbatch. Second, we recursively generate the microbatch configurations according to the cost model. Specifically, balancing can be done by looking ahead all tokens to be chunked in a recursive manner, as shown in Figure 10. The initial batch contains a single microbatch with all tokens, which is then recursively split into two cost-balanced microbatches until it reaches a balanced setup.

Figure 11 shows the detailed pseudocode. The algorithm complexity is $O(\log L)$ so it can be quickly solved online. For simplicity, we omit the details of split, which divides requests in a batch into chunks and returns a new microbatch set whose aggregated cost is equal to the objective ($0.5 \times cost(b)$). This ensures that each microbatch has sufficient tokens to fully utilize the GPU. One thing to note is that the generation halts once the number of tokens to form a batch is below a threshold (line 4–5).

A key to the effectiveness of the above algorithm is to accurately estimate the execution time (i.e., cost) of a microbatch. We derive the cost model using a bottom-up approach: we first model the cost of executing a chunk of a request, then we sum the cost of all chunks in a microbatch as its cost. Specifically, suppose we have a microbatch set $\mathcal{B}$, denoted by $\mathcal{B} = \{b_1, b_2, \ldots, b_m\}$, The chunks are chunked from a request set of size n, denoted by $\mathcal{R} = \{r_1, r_2, \ldots, r_n\}$. The cost of a chunk c_{ij}, $\text{cost}_{c_{ij}}$, can be formulated as follows:

$$\text{cost}_{c_{ij}} = \alpha \left(\overbrace{p_{ij} c_{ij}}^{\text{prefix-attn}} + \overbrace{\frac{c_{ij}^2 + c_{ij}}{2}}^{\text{self-attn}} \right) + \beta \overbrace{c_{ij}}^{\text{FFN}} + \overbrace{\gamma}^{\text{other}} \quad (1)$$

The equation consists of four parts: the cost to compute attention with previous tokens (**prefix-attn**); the cost to compute attention with the chunk itself (**self-attn**); the cost of computing the activations (**FFN** (Feed-Forward Network)) for tokens; and others. The prefix tokens of each chunk can be calculated as $p_{ij} = \sum_{k=1}^{j-1} c_{ik}$. The **prefix-attn** and **self-attn** models the quadratic cost of attention computation missed by existing models, e.g., NanoFlow [56] does not consider **self-attn**, while DistServe [55] does not take **prefix-attn** into account.

Our model depends on several hyperparameters (e.g., α) that can be determined through offline profiling: before the system is deployed for serving, we run multiple inference samples offline, collect their execution times, and then use the least squares method [49] to determine all hyperparameters.

Given the cost of each chunk, we can sum all the costs of chunks in a microbatch to get the cost of the microbatch:

$$b_k = \left\{ c_{ij} \mid x_{ij} = k \wedge c_{ij} > 0 \right\}, \quad \forall k \in \{1, \ldots, m\} \quad (2)$$

$$\text{cost}_{b_k} = \sum_{\substack{(i,j) \\ x_{ij}=k}} \text{cost}_{c_{ij}} - (|b_k| - 1)\gamma \quad (3)$$

Note that the term $-(|b_k| - 1)\lambda$ reflects the elimination of duplicated parameter-loading when executing a batch, as requests in a batch share the same model parameter. Like other hyperparameters, λ can be fitted with offline profiling.

Empirically, our cost model accurately models the execution time of a microbatch for common sequence lengths in Figure 15. As a result, the pipelined execution with our lookahead formulation can significantly reduce the execution bubbles (see Figure 14).

Discussion: the generality of lookahead batch formulation and cost model. While in principle, we could also apply lookahead batch formulation to general LLM serving with pipeline execution, it has one obstacle that the formulation assumes a sufficient number of requests queued to "lookahead" to be effective. Under normal serving without bursts, waiting

for requests to be looked ahead may add additional latency, which we leave possible solutions as a future work.

Besides, readers may findEq. 1 still has a part that has a linear correlation with the number of tokens (FFN), so if the cost is dominated by FFN, existing token-count-based cost models may suffice. We argue that our retrofitted cost model is still important because the quadratic terms (prefix-attn and self-attn) would become significant when the token count increases (e.g., for requests with more than 4K tokens, which are common in real-world workloads [15], see §5.1), so existing works can leverage our model for a more accurate estimation of microbatch execution time.

4.4 Dynamic Restore and Fault Tolerance

Dynamic parameter restoration. While dynamic parameter drop described in §4.1 can free up memory for new requests under memory overloading, the pipelined execution is not optimal under normal execution because (1) pipelined execution suffers from more frequent weight loading and (2) it has bubbles. Normal execution cannot simply apply our lookahead scheduling described in §4.3 because there are insufficient numbers of requests to balance.

To this end, KUNSERVE dynamically restores parameters to return to a normal non-pipelined execution once the overloading fades away. Specifically, when the monitor detects that the total KVCache usage is below a threshold, it triggers a restoration process by loading the dropped parameters back to the GPUs. Currently we use a simple threshold where the memory usage is below 50 % of the GPU (without drop). The missing parameters are pulled from instances whenever possible using the network between instances.

Two things need to be noted about the restoration. First, we overlap restoring with the normal request processing. Second, since KUNSERVE is concurrently restoring when the request is executing, the parameter pulling process may block activation transfer of normal requests, causing latency increases (see Figure 14). Thus, we adopted a similar coordinated network transfer approach described in §4.2 to ensure a smooth execution of pipelined requests by prioritizing the pipeline network over the parameter transfer.

Fault tolerance. Unlike traditional LLM serving where failures between instances are isolated, a failure node in KUNSERVE can disrupt other instances that are involved in the same pipeline-parallel group. Thus, we dynamically restore these affected instances to ensure normal execution under failures. By replicating parameters in host DRAM or SSDs, we can always ensure successful parameter restoration.

5 Evaluation

5.1 Experiment setup

Testbed. We evaluate KUNSERVE on two clusters listed in Table 2. Cluster A has one GPU per server so it is typically used for running small models (e.g., 14 B models). Cluster B has multiple GPUs per server interconnected with fast NVLink, so it is suitable for running larger models (e.g., 72 B models) with tensor parallelism.

Evaluated models. Similar to prior works [8, 38, 55], we choose open-source models with leading accuracy: Qwen-2.5-14B and Qwen-2.5-72B [45]. Both models adopt GQA [11] to reduce KVCache size while maintaining high accuracy. We do not choose models with huge KVCache usage (e.g., models with MHA [46]) that could easily exhaust GPU memory—though KUNSERVE is more effective when serving such models. This is because these models are being replaced by more KVCache-efficient variants. Table 1 lists instance configurations of each model. For the 72B model, we use tensor parallelism to serve requests on multiple GPUs.

Evaluated traces and datasets. Since memory overloading is sensitive to the request arrival pattern, we use a real-world trace BurstGPT [48] with known request arrival information (i.e., the invocation time of each request) as our main evaluated application. Following the guide of BurstGPT, we scale BurstGPT's RPS to fit the serving capacity of our testbed using a scaling method that preserves the temporal pattern of the trace. Specifically, we upscale the trace with TraceUpscaler [41], and ensure that the average memory demand is lower than 60% of the total memory during the entire evaluation of the trace.

Besides the arrival pattern, LLM serving is also sensitive to the input and output length of requests. Thus, given the trace, we further evaluate requests from representative datasets representing different scenarios, similar to prior works [32, 34]:

- **BurstGPT.** It is the original dataset of BurstGPT [48], representing a conversion workload so both TTFT and TPOT are important. The average input and output lengths are 642 and 262, respectively.
- **ShareGPT.** ShareGPT [3] is another popular chatbot dataset that is widely evaluated on [8, 44, 51, 55]. Its input and output lengths are longer than BurstGPT, representing a workload that is more sensitive to GPU memory provisioning. The maximal input length is 4K, and the average input and output lengths are 1,660 and 373, respectively. Like BurstGPT, low TTFT and TPOT are both important for benchmark using this dataset.
- **LongBench.** LongBench [15] is another popular dataset used for evaluating document summarization tasks [55], e.g., summarizing news, articles and scientific papers. The average input length is 5.9 K and the average output length

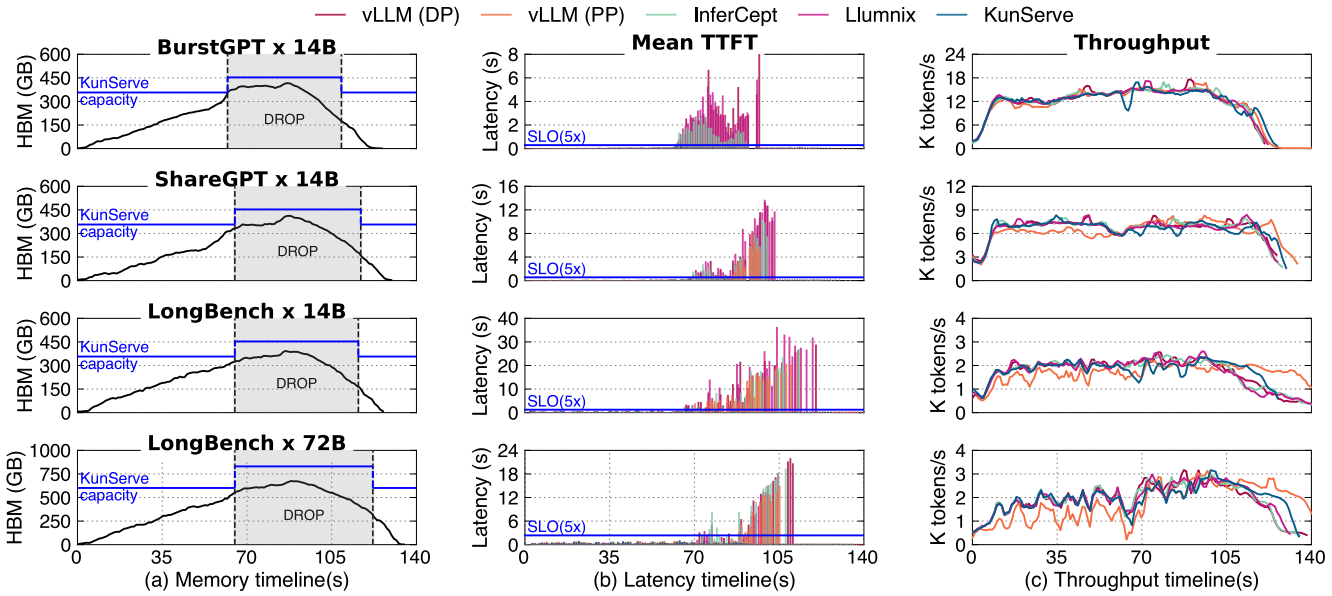

Figure 12: First column: the memory usage pattern of KUNSERVE. *Second column: the mean TTFT during the evaluation. Third column: the throughput during the evaluation.*

is 499. Since the user expects a quick response to the summarized content, TTFT is also important.

Baselines. We compared with the state-of-the-art LLM serving systems with various techniques to cope with memory overloading. For all systems, we have carefully tuned their configurations to meet the optimal performance without memory overloading. We have also enabled all known serving optimizations to these systems even though the vanilla systems are not optimized (e.g., InferCept [7]). For those with our optimizations, we have calibrated that our optimizations enabled better performance than the original open-sourced codebase. More specifically, our baselines are:

- **vLLM (default + PP) [30].** We compare two configurations of vLLM (release v0.6.3): The default configuration stores the entire parameters on each instance, while pipelined parallelism (PP) further frees half of the parameters on each instance and leverages PP to execute requests across two instances. This setup frees up more memory for KVCache, but it also introduces pipelined execution overhead. By default, vLLM uses recomputation to cope with memory overloading. We compared the vLLM with swapping to InferCept described below.
  Before the evaluation, we carefully tuned the configurations of vLLM. Specifically, we tuned the block size to achieve the best performance under our setup. We chose 64 because (1) it is small enough to avoid memory fragmentation while (2) it is sufficiently large to achieve good performance [21].

Table 2: Testbed configurations. s and g denote the number of servers and GPUs per host, respectively. Bandwidth (unidirectional) is reported for both networks.

	Cluster A ($s \times g$)	**Cluster B** ($s \times g$)
GPU	A800 80 GB (8×1)	H800 80 GB (2×8)
Scale-up Network (GPU-GPU)	N/A	300 GB/s NVLink
Scale-out Network (GPU-GPU)	200 Gbps RDMA	400 Gbps RDMA

- **InferCept [7].** InferCept designs an optimized swap mechanism that eliminates IO idle time atop vLLM. We tried to compare its original open-sourced version, but found its performance is 1.2–5.1 × slower in TTFT and 1.2–1.9 × in TPOT than the chosen vLLM release even without memory overloading. This is because it was implemented on an old version of vLLM (v0.2.0), where important optimizations (e.g., FlashAttention/FlashInfer kernels [17, 21], chunked prefill [8]) are missing. Therefore, we integrated our scheduler and attention backend into the original InferCept for a fair comparison.
- **Llumnix [44].** Llumnix adopts load balancing to cope with memory overloading of an instance, and migrates KVCache between instances to free sufficient memory in case of insufficient memory even with load balancing. We compared with the latest version of Llumnix (release v0.1.0).

5.2 End-to-end Results

End-to-end serving performance. We first measure the end-to-end latency of serving requests when running BurstGPT

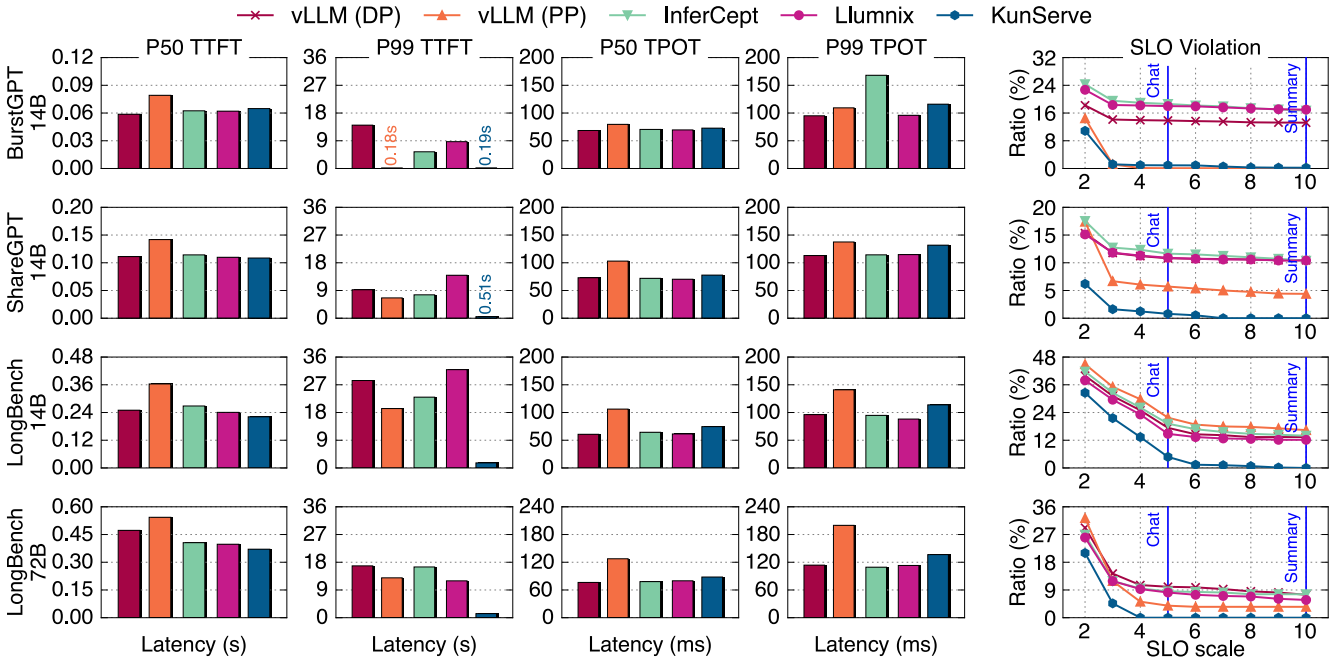

Figure 13: The end-to-end latency results. Column from 1 to 4 is the end-to-end metrics of different workloads. The last column is the SLO violation of TTFT and TPOT with different SLO scales.

with different datasets on different systems, where the latency is measured from the client's perspective, i.e., the time from the client sending a request to receiving the tokens.

The second column of Figure 12 presents how the mean TTFT changes over time given a measured time window (e.g., 100s), and Figure 13 presents the zoomed-in view of the P50 and P99 latencies when evaluating different workloads on different models. First, KunServe has 12.7–72.2× faster P99 TTFT than other baselines, because it frees up sufficient memory under memory overloading, which enables requests queued in other systems to be served with a larger batch size. For other systems, they either suffer from recomputation overhead (vLLM), or queuing overhead waiting for swapping (InferCept) or migration (Llumnix) under memory overloading. Specifically, the timeline plotted in Figure 12 clearly shows that the TTFT increase coincides with the increased KVCache demands (the first column in Figure 12).

Although vLLM (PP) has a larger KVCache capacity, it still suffers from medium and tail latency increases due to the lower throughput. As shown in the third column of Figure 12, the average throughput of PP is 3.3–21.8% slower than other systems, because PP has bubbles during execution. Such a lower throughput leads to more KVCache capacity being required under bursts since pending requests are not digested by the system. Meanwhile, unlike KunServe that schedules pending requests to eliminate bubbles, vanilla pipelined execution cannot simply adopt lookahead batch formulation techniques (§4.3) because it requires waiting for sufficient requests to be scheduled. Such waiting also leads to increased end-to-end latencies.

Compared to other baselines, KunServe trades a little increase in P50 TPOT, and P99 TPOT because it executes requests in a larger batch to eliminate queuing. For example, in LongBench-14B workload, the P50 TPOT of KunServe is 15.8–22.7% higher than other baselines. We believe it is a reasonable trade-off because such increases are still within the SLOs of targeted applications, which we describe next. Interestingly, KunServe even has a little P50 TTFT improvement in the LongBench workload. This is because the long and diverse input of requests in this workload makes the system more prone to memory overloading caused by severe memory fragmentation [44]. Thus, the many queued requests affect normal requests.

SLO attainment. SLO is an important metric for serving systems [32, 55], which defines the maximum acceptable latency for a request. Requests whose latency exceeds the SLO are not useful because users may abandon them [40]. Because different applications have different maximum acceptable latency requirements (SLOs), we evaluate the SLO violation of all systems under different SLO scale factors, similar to previous works [32, 38, 43, 55].

Specifically, the last column of Figure 13 shows the SLO violation of all systems with different SLO scale factors, where a scale factor of N means that the maximal tolerable latency

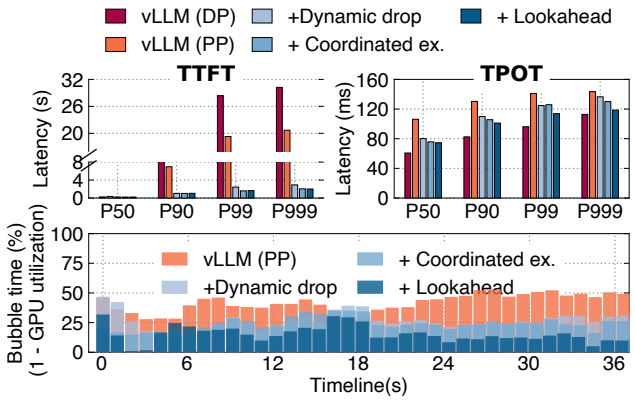

Figure 14: An ablation study of running KUNSERVE *on Qwen-2.5-14B on LongBench dataset. A smaller bubble time directly implies a better GPU utilization.*

is N times the P50 latency of the best baseline. To help understand how the reduced SLO violations of KUNSERVE benefit end-to-end applications, we also mark the typical scale for our evaluated applications, i.e., we set 5 for chat—a tight SLO as it requires quick responsiveness, while for document summarization, we set a looser factor of 10, following previous works [55]. We can see that KUNSERVE achieves 7.2—12.8% average SLO violation reductions on various workloads, and more importantly, it almost eliminates all violations with a scale larger than 4 for all workloads. Other baselines cannot eliminate SLO violations even with an extremely loose factor of 10 because during bursts, there are considerable numbers of queued requests suffering from 45—840× tail latency increases.

Multi-GPU instance performance. Due to space limitations, we only present the results of the model (Qwen-2.5-72B) that requires multi-GPU for serving on the LongBench dataset. Results on other datasets are similar. As shown in Figure 12 and Figure 13, the trend is similar to that of single-GPU instances: KUNSERVE reduces the P99 latency by 8.4–11.9× compared to other baselines, at the cost of a slight (18.3–22.7%) increase in P50 TPOT and P99 TPOT. The multi-GPU model achieves similar results because each instance (containing multiple GPUs) can be viewed as a whole as a single logical GPU. The multi-GPU model benefits even more from dropping parameters because the relative ratio of parameter memory is large, as shown in Table 1.

5.3 Ablation Studies

To study the effectiveness of each technique proposed in §4, we conducted an ablation study on the system performance with different techniques incrementally enabled. Figure 14 shows the detailed study results on the LongBench dataset with Qwen-2.5-14B model. We omit other workloads and models due to space limitation since they have similar results.

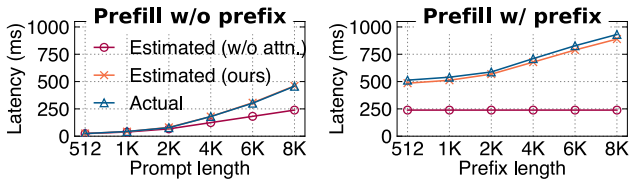

Figure 15: A comparison of execution latency estimated with our cost model and the real execution time of a Qwen-2.5-14B model in A800 GPUs. Left: the execution without prefix attention while right: the execution with prefix attention.

We report the end-to-end request latencies during the burst period in Figure 12.

Effectiveness of dynamic parameter drop. First, we can see that parameter drop contributes (+Dynamic drop) to the most tail latency reductions. On the LongBench workload, the P90, P99 and P999 TTFT of KUNSERVE are reduced by 8.8×, 11.7× and 10.3× compared to vLLM (DP). The key reason is that it completely eliminates queuing delays. Specifically, under bursts, there are 87 queued requests (whose TTFT > SLO(5×)) in this evaluation, KUNSERVE executes them with enlarged GPU memory freed by dropping parameters. Though a larger batch size and pipeline bubbles lead to a TPOT increase in request processing (21–31.9% increase compared to the original DP scheduling), it is still orders of magnitude smaller than queuing introduced by insufficient memory of vanilla vLLM.

Effectiveness of coordinated exchange. Second, with coordinated exchange (+ Coordinated ex.), KUNSERVE further reduces the P99 and P999 TTFT by 1.5×, and 1.4× respectively. Meanwhile, it reduces the P90 and P999 TPOT by 5%. Coordinated exchange benefits both the TTFT and TPOT because without it, the prefill of new requests as well as their decode requests cannot execute smoothly, because the intermediate activation suffers significant stalls due to exchanging the KVCache. Since the exchange time (1.3s) is larger than the typical execution time (e.g., 221ms for prefill and 60ms for decode), the stall is non-trivial.

Effectiveness of lookahead batch formulation. With lookahead batch formulation (+ Lookahead), we further reduce the P90, P99, and P999 TPOT by 4.5%, 10.6%, and 9.7%, respectively. The reduction in latency directly comes from the more efficient pipeline execution: without lookahead batch formulation, KUNSERVE suffers 21.9% bubble time (the ratio of idle GPU cycles) on average during pipelined execution, while with it, the bubble time is only 8.3%. The reduced bubble time further improves throughput by 20%.

5.4 Batch Formulation Cost Model Accuracy

To evaluate the accuracy of KUNSERVE's cost model described in §4.3, we compare it with a baseline cost model

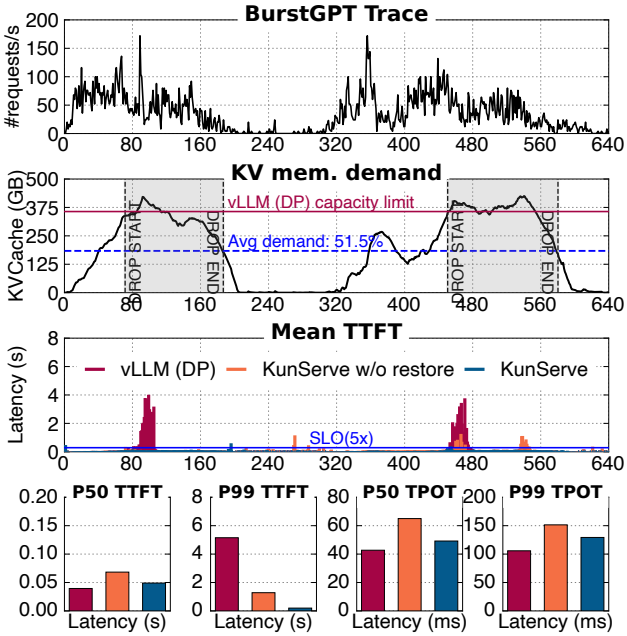

Figure 16: The performance of KUNSERVE *and its baselines in a long run (640s) of BurstGPT.*

neglecting attention computation cost found in existing work [56] and the ground truth. To demonstrate the generality of our model in both prefill and chunked prefill, we evaluate both requests without attention chunk ($R4^0$ in Figure 9 (c)) and with it ($R4^1$). As shown in Figure 15, for both cases, our cost model shows less than 5% deviation while the current formulation without considering attention has up to 48% and 74% deviations for requests without and with prefix attention, respectively. This confirms the importance of considering the attention computation cost in the cost model.

5.5 Effectiveness of Dynamic Restoration

To show the effectiveness of dynamic parameter restoration, Figure 16 presents the serving performance over a long run of BurstGPT workload with multiple overloading periods. To help understand the behavior of KUNSERVE, we mark the time periods with dropping as grey boxes, other periods are running without parameter drop.

First, we observe that dynamic parameter restoration reduces the P50 latencies of TTFT and TPOT by 28 % and 23 %, respectively, due to the reduction of unnecessary pipeline execution. Second, restoration improves the P99 TTFT and TPOT by 6.4 × and 1.2 ×, respectively. Without restoration, KUNSERVE falls back to vLLM (PP), resulting in lower throughput during normal periods, and consequently suffers from larger bursts with insufficient memory even with the dropped parameters, as illustrated at the beginning of the second wave in the third row of Figure 16 (time 440s).

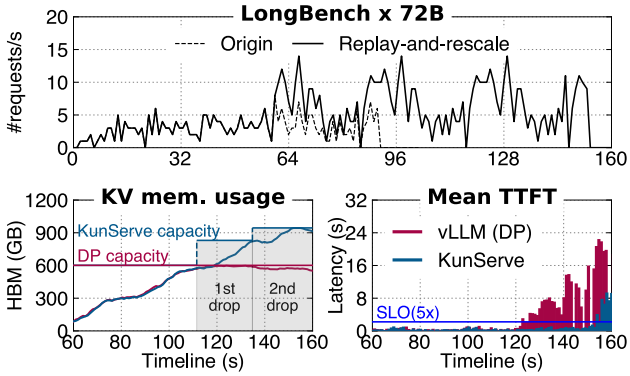

Figure 17: An evaluation of KUNSERVE *running Qwen-2.5-72B under extreme bursts.*

5.6 Performance under Extreme Bursts

While KUNSERVE drops parameters to mitigate queuing, the memory that can be freed is bounded by the model size (see Table 1), so we have a limit in handling overloading caused by bursts. Nevertheless, KUNSERVE can handle bursts much longer than existing systems, i.e., longer than any burst we have seen in the BurstGPT trace.

To evaluate the limit in handling bursts with KUNSERVE, Figure 17 shows the performance of KUNSERVE and vLLM when running under an unrealistic extreme burst. Specifically, to evaluate an extreme burst, we use a BurstGPT setup as follows: upon meeting the first burst, we repeatedly replay the bursts until all evaluating systems are out of memory. The setup is shown in the first row of Figure 17 while the second row compares the performance of KUNSERVE and vLLM (DP). The evaluated model is Qwen-2.5-72B. First, KUNSERVE reaches the memory limit in 152s, which is 1.5 × longer (starting from 60s) than vLLM thanks to the dropped memory. During this period, KUNSERVE triggers 2 times of parameter dropping, resulting in 57% incrementally freed KVCache memory. Before KUNSERVE reaches the memory limit, KUNSERVE meets no SLO (5 ×) violations while vLLM suffers up to 42 × TTFT increase.

While KUNSERVE also suffers from latency increases when out of memory, we don't encounter such a situation under real-world traces. More importantly, the much longer standing time of KUNSERVE allows the serving systems to smoothly scale up new instances to handle the bursts.

6 Discussion

Supporting MoE models. Our current implementation focuses on dense models, while Mixture of Experts (MoE) models are becoming increasingly popular recently: A key feature is that the inference of a request only activates a small subset of model parameters. For common MoE serving configurations like expert parallelism (EP) [20], KUNSERVE

seamlessly supports them because EP only changes the memory layout within an instance, while KUNSERVE focuses on managing the GPU memory across serving instances. More importantly, KUNSERVE is still effective for MoE models because KUNSERVE only relies on the assumption that the model weights occupy a large portion of the GPU memory on an instance, which holds even with a sparse activation of experts (see Table 1). The assumption holds because though a request only requires a small portion of the model parameters, an instance still needs to load all the (large) model parameters to handle batches of requests that may activate all the experts.

Compatibility with different parallelism. KUNSERVE is compatible with different parallelism in LLM serving. KUNSERVE only changes the parameter layout across instances in layer granularity, which is orthogonal to both the intra-layer layout change (e.g., EP and TP) within one instance and instance cooperation in SP. Thanks to the LLM's modular structure, the intra-instance and inter-instance parallelism techniques can be applied together [42, 51].

Comparison with autoscaling. Autoscaling—adding more instances to handle overloads—is also a common approach to handle memory overloading [53]. A key difference is that KUNSERVE does not have the cold start time—the time to make an instance capable of serving. Thus, KUNSERVE is better than autoscaling in cases where dropping alone is able to handle the overloading, as the cold start time is typically non-trivial for LLM providers [40]. Nevertheless, for long bursts (see §5.6), KUNSERVE still incorporates autoscaling since the memory that can be freed by dropping is limited: the continuously coming requests from the burst will exhaust all the free memory freed by dropping.

7 Related Work

Handling memory overloading with lossy methods. One possible way to handle memory overloading is to reduce the memory footprint of the serving, e.g., by compressing the activations [19, 31]. For example, FP8 quantization [47] reduces the token memory usage by 2×, and methods like SparseGPT [22] prune parameters to 50% sparsity. Unfortunately, such methods are lossy and can lead to model accuracy degradation or compromised user experience [33]. KUNSERVE copes with the performance degradation caused by memory overloading without sacrificing the model accuracy.

Handling memory overloading with lossless methods. KUNSERVE continues the line of work on handling memory overloading during LLM serving without modifying the model inference [7, 30, 30, 40, 44, 50, 55]. These works focus on allowing queued requests to execute by reorganizing GPU memory either with swap or migration-based methods, which do not create more space for execution so they either sacrifice ongoing requests or queued requests, as analyzed in §2.3. In contrast, KUNSERVE frees more memory for execution with a new parameter-centric memory management method.

LLM serving optimizations. Considerable research has focused on improving the efficiency of LLM serving under abundant memory [9, 17, 18, 27, 30, 38, 39, 55]. KUNSERVE builds on these works and seamlessly integrates with them. A recent work—POD-ATTENTION [29]—proposes a better chunked prefill implementation. It is orthogonal to our work and KUNSERVE can benefit from its high-performance kernel to get better performance in all states. NanoFlow [56] provides us with a more efficient microbatch scheduling, which is of help to KUNSERVE after parameter dropping.

OS techniques for handling memory overloading. Handling memory overloading has been studied in operating systems for decades: e.g., Linux adopted a swap-based mechanism to handle memory pressure [1]. KUNSERVE leverages the domain-specific knowledge of LLM serving to expose more memory to serving requests beyond the limit of a general-purpose swap-based method.

8 Conclusion

In this paper, we are the first to demonstrate that parameter-centric memory management can effectively address the latency spikes caused by memory overloading in LLM serving. We built KUNSERVE, an LLM serving system that cooperatively drops parameters to free up memory to eliminate queuing under overloading. We also proposed a set of techniques to ensure all requests execute efficiently after parameter dropping, including drop plan generation with local unified memory management, coordinated KVCache exchange and lookahead batch formulation. KUNSERVE reduces tail TTFT by up to 72.2× compared to state-of-the-art systems like Llumnix, vLLM and InferCept.

9 Acknowledgement

We sincerely thank our shepherd Jayashree Mohan and the reviewers from OSDI'25 and EuroSys'26 for their insightful feedback. We are grateful to Wencong Xiao from ByteDance, Mingcong Han, Hanze Zhang, Xian Xu, Yu Xia, Yingyi Hao, and Hongrui Xie from IPADS for their valuable advice. We also thank the ByteDance Seed-Infra team for their platform support. We thank Chao Fei from KAUST for his contributions to the codebase of KUNSERVE. This work was supported in part by the National Natural Science Foundation of China (No. 62572302 and 62272291), and the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (JYB2025XDXM122).

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] Sharegpt_gpt4, 2024. https://huggingface.co/datasets/shibing624/sharegpt_gpt4, 2024.

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

[5] How multi-node inference works for massive llms like deepseek-r1. https://www.baseten.co/blog/how-multi-node-inference-works-llms-deepseek-r1/#from-single-node-to-multi-node-infrastructure, 2025.

[6] Lower latency and higher throughput with multi-node deepseek deployment. https://www.perplexity.ai/hub/blog/lower-latency-and-higher-throughput-with-multi-node-deepseek-deployment, 2025.

[7] ABHYANKAR, R., HE, Z., SRIVATSA, V., ZHANG, H., AND ZHANG, Y. Infercept: Efficient intercept support for augmented large language model inference. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024* (2024), OpenReview.net.

[8] 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.

[9] 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).

[10] 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.

[11] 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.

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

[13] 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.

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

[15] BAI, Y., LV, X., ZHANG, J., LYU, H., TANG, J., HUANG, Z., DU, Z., LIU, X., ZENG, A., HOU, L., DONG, Y., TANG, J., AND LI, J. Longbench: A bilingual, multitask benchmark for long context understanding. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024* (2024), L. Ku, A. Martins, and V. Srikumar, Eds., Association for Computational Linguistics, pp. 3119–3137.

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

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

[18] 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).

[19] DEEPCHECKS. Top llm quantization methods and their impact on model quality, 2024. https://www.deepchecks.com/top-llm-quantization-methods-impact-on-model-quality/, 2024.

[20] DEEPSEEK-AI, LIU, A., FENG, B., XUE, B., WANG, B., WU, B., LU, C., ZHAO, C., DENG, C., ZHANG, C., RUAN, C., DAI, D., GUO, D., YANG, D., CHEN, D., JI, D., LI, E., LIN, F., DAI, F., LUO, F., HAO, G., CHEN, G., LI, G., ZHANG, H., BAO, H., XU, H., WANG, H., ZHANG, H., DING, H., XIN, H., GAO, H., LI, H., QU, H., CAI, J. L., LIANG, J., GUO, J., NI, J., LI, J., WANG, J., CHEN, J., CHEN, J., YUAN, J., QIU, J., LI, J., SONG, J., DONG, K., HU, K., GAO, K., GUAN, K., HUANG, K., YU, K., WANG, L., ZHANG, L., XU, L., XIA, L., ZHAO, L., WANG, L., ZHANG, L., LI, M., WANG, M., ZHANG, M., ZHANG, M., TANG, M., LI, M., TIAN, N., HUANG, P., WANG, P., ZHANG, P., WANG, Q., ZHU, Q., CHEN, Q., DU, Q., CHEN, R. J., JIN, R. L., GE, R., ZHANG, R., PAN, R., WANG, R., XU, R., ZHANG, R., CHEN, R., LI, S. S., LU, S., ZHOU, S., CHEN, S., WU, S., YE, S., MA, S., WANG, S., ZHOU, S., YU, S., ZHOU, S., PAN, S., WANG, T., YUN, T., PEI, T., SUN, T., XIAO, W. L., AND ZENG, W. Deepseek-v3 technical report. *CoRR abs/2412.19437* (2024).

[21] FLASHINFER AI. Flashinfer: Kernel library for llm serving. https://github.com/flashinfer-ai/flashinfer, 2024.

[22] FRANTAR, E., AND ALISTARH, D. Sparsegpt: Massive language models can be accurately pruned in one-shot. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA* (2023), A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, Eds., vol. 202 of *Proceedings of Machine Learning Research*, PMLR, pp. 10323–10337.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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).

[28] 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).

[29] KAMATH, A. K., PRABHU, R., MOHAN, J., PETER, S., RAMJEE, R., AND PANWAR, A. Pod-attention: Unlocking full prefill-decode overlap for faster LLM inference. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2025, Rotterdam, Netherlands, 30 March 2025 - 3 April 2025* (2025), L. Eeckhout, G. Smaragdakis, K. Liang, A. Sampson, M. A. Kim, and C. J. Rossbach, Eds., ACM, pp. 897–912.

[30] 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.

[31] LI, S., NING, X., WANG, L., LIU, T., SHI, X., YAN, S., DAI, G., YANG, H., AND WANG, Y. Evaluating quantized large language models. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024* (2024), OpenReview.net.

[32] 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.

[33] MARCHISIO, K., DASH, S., CHEN, H., AUMILLER, D., ÜSTÜN, A., HOOKER, S., AND RUDER, S. How does quantization affect multilingual llms? In *Findings of the Association for Computational Linguistics: EMNLP 2024, Miami, Florida, USA, November 12-16, 2024* (2024), Y. Al-Onaizan, M. Bansal, and Y. Chen, Eds., Association for Computational Linguistics, pp. 15928–15947.

[34] 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.

[35] NVIDIA. Nvidia dgx superpod: Next generation scalable infrastructure for ai leadership. https://docs.nvidia.com/dgx-superpod/reference-architecture/scalable-infrastructure-h200/latest/_downloads/bbd08041e98eb913619944ead1f92373/RA-11336-001-DSPH200-ReferenceArch.pdf#page=8.10, 2024.

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

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

[38] PATEL, P., CHOUKSE, E., ZHANG, C., SHAH, A., GOIRI, Í., MALEKI, S., AND BIANCHINI, R. Splitwise: Efficient generative LLM inference using phase splitting. In *51st ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2024, Buenos Aires, Argentina, June 29 - July 3, 2024* (2024), IEEE, pp. 118–132.

[39] 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).

[40] 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).

[41] SAJAL, S. M., ZHU, T., URGAONKAR, B., AND SEN, S. Traceupscaler: Upscaling traces to evaluate systems at high load. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024* (2024), ACM, pp. 942–961.

[42] 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).

[43] 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).

[44] 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.

[45] TEAM, Q. Qwen2.5: A party of foundation models, September 2024.

[46] 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.

[47] VLLM PROJECT. Llm compressor, 2025. https://github.com/vllm-project/llm-compressor, 2025.

[48] 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.

[49] WEISSTEIN, E. W. "least squares fitting." from mathworld–a wolfram resource. https://mathworld.wolfram.com/LeastSquaresFitting.html, 2025.

[50] 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).

[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. 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.

[52] WU, B., ZHONG, Y., ZHANG, Z., HUANG, G., LIU, X., AND JIN, X. Fast distributed inference serving for large language models. *CoRR abs/2305.05920* (2023).

[53] ZHANG, D., WANG, H., LIU, Y., WEI, X., SHAN, Y., CHEN, R., AND CHEN, H. Blitzscale: Fast and live large model autoscaling with O(1) host caching. In *19th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2025, Boston, MA, USA, July 7-9, 2025* (2025), L. Zhou and Y. Zhou, Eds., USENIX Association, pp. 275–293.

[54] 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.

[55] 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.

[56] ZHU, K., ZHAO, Y., ZHAO, L., ZUO, G., GU, Y., XIE, D., GAO, Y., XU, Q., TANG, T., YE, Z., KAMAHORI, K., LIN, C., WANG, S.,

KRISHNAMURTHY, A., AND KASIKCI, B. Nanoflow: Towards optimal large language model serving throughput. *CoRR abs/2408.12757* (2024).