MELL: Memory-Efficient Large Language Model Serving via Multi-GPU KV Cache Management

Qianli Liu[1], Zicong Hong[1], Peng Li[2], Fahao Chen[3] and Song Guo[1]
[1]Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong
[2]School of Cyber Science and Engineering, Xi'an Jiaotong University, China
[3]School of Computer Science and Engineering, University of Aizu, Japan
qianli.liu@connect.ust.hk, ziconghong@gmail.com, pengli@xjtu.edu.cn, chenfh@ieee.org, songguo@cse.ust.hk

***Abstract*—Serving large language models (LLMs) for massive users is challenged by the significant memory footprint of the transient state, known as the *key-value (KV) cache*, which scales with sequence length and number of requests. Instead of renting or buying more expensive GPUs, the load imbalance of the KV cache across GPUs, coupled with recent advances in inter-GPU communication, provides an opportunity to serve more requests via request migration. However, high migration overhead and unpredictable request patterns make it challenging. Therefore, this paper proposes MELL, a memory-efficient LLM serving system via *multi-GPU KV cache management*. It saves the number of GPUs needed in the system by considering the dynamic KV cache load and the costly request migration. Specifically, we first develop an adaptive request migration mechanism to balance the computational and communication overheads and adapt to diverse resource conditions. Then, we design an online algorithm tailored to a multi-LLM request and multi-GPU scheduling problem with migration enabled. It aims to minimise the required GPUs while limiting the number of migrations. Finally, we implement a prototype of MELL and demonstrate that it reduces the number of GPUs by 31% and increases the GPU utilization by 43% at most compared to existing LLM serving systems.**

***Index Terms*—large language model serving, key-value cache.**

I. INTRODUCTION

The capability of Large Language Models (LLMs) [1]–[3] to understand and produce human-like text has established them as a central component of AI, dramatically improving many complex language-related tasks across industries. As the use of LLMs becomes more widespread, it is essential to deploy them on GPU clusters with a large number of GPUs [4] and provide users with seamless access [5]–[9].

To improve the LLM inference efficiency on GPUs, *key-value (KV) cache* is one of the most critical modules [10]. It stores the keys and values of all previous tokens in GPU memory for each LLM inference to avoid redundant and repeated computations. Despite this advantage, there is a problem with GPU memory during long context processing and generation. Unlike the model weights, the KV cache is subject to size growth due to sequence length and batch size. As the demand for longer sequence lengths (along with larger batch sizes) grows [11], the KV cache size problem becomes more pronounced. Statistics show that the KV cache now often consumes over 30% of the GPU memory [12].

Memory management is important for accommodating more KV cache without renting or buying more expensive GPUs. Existing KV cache management works either compress the KV cache [13]–[18] or offload the KV cache to CPU memory [19]–[22]. However, the former inevitably degrades LLM performance through quantization or sparsity, while the IO bottleneck between CPU memory and GPU limits the latter.

To avoid these problems, we observe that the size of the KV caches on each GPU varies over time. Particularly, some GPUs are overwhelmed by the growth of the KV cache from running requests, while others have a lot of unused memory because the KV cache is released for completed requests. This motivates us to schedule LLM requests with their KV cache from a heavily loaded GPU to a less loaded GPU to avoid renting or buying a new GPU. According to our preliminary experiments (see **Finding 3** in § III), such migration allows an LLM serving system to handle at most 60% more LLM requests than that without migration. A few works have demonstrated the feasibility of migrating requests across GPUs without significant service halt (i.e., live migration), e.g., Llumnix [23] and ServerlessLLM [24].

However, there are three challenges that need to be addressed regarding scheduling. **1) Unpredictable request patterns**: Besides the arrival time, the processing time of requests is difficult to learn due to the unpredictability of LLM response length. Moreover, resource utilization changes as requests are processed due to updates in the KV cache. **2) High migration overhead**: Existing migration is either compute-intensive [24] or communication-intensive [23] due to the KV cache transfer or re-prefill, respectively. Thus, the scheduling needs to balance the computational and communication overhead caused by migration. **3) Theoretical guarantee**: most of the existing scheduling is based on a heuristic design (e.g., load swapping between GPUs with lowest load and highest load

This research was supported by fundings from the Hong Kong RGC General Research Fund (152244/21E, 152169/22E, 152228/23E, 162161/24E), Research Impact Fund (No. R5011-23, No. R5060-19), Collaborative Research Fund (No. C1042-23GF), Theme-based Research Scheme (T43-518/24-N), National Natural Science Foundation of China (No. 62471383), Areas of Excellence Scheme (AoE/E-601/22-R), and Hong Kong Generative AI Research and Development Center from InnoHK. Corresponding authors: Zicong Hong, Song Guo.

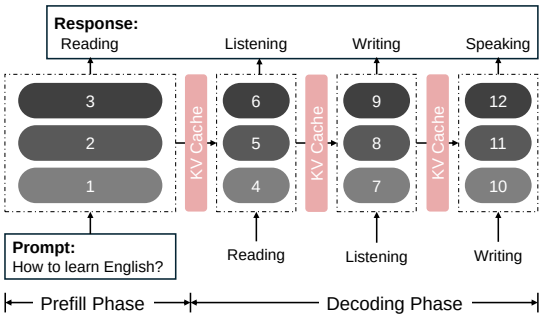


Fig. 1: Serving procedure of an LLM request.

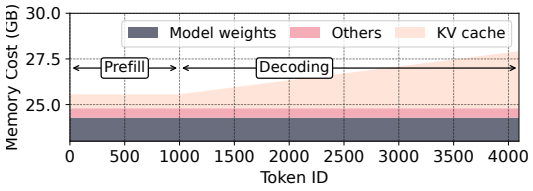


Fig. 2: The memory cost of processing a request with 4096 tokens on LLaMA-13B.

repeatedly [23]) without a theoretical performance guarantee.

To solve these challenges, this paper proposes MELL, a memory-efficient LLM serving system enabled by a novel multi-GPU KV cache management. It perceives the system's dynamic KV cache load and resources, decides on the placement of LLM requests during their processing, and efficiently migrates the requests' KV cache to save the number of GPUs.

We summarize our contribution as follows.

- We design an adaptive request migration mechanism by switching in real-time between the token and KV cache migration to balance computational and communication overheads for the dynamic environment.
- We develop an online KV cache scheduling algorithm in a multi-request multi-GPU environment to minimize the number of GPUs needed and limit the number of migrations. It has been rigorously proved to have a competitive ratio with the optimal solution of $4/3$ at most.
- We implement a prototype of MELL and demonstrate that it substantially reduces the number of GPUs by $9\% \sim 31\%$ and increases the GPU utilization by $10\% \sim 43\%$ compared to the existing LLM serving systems.

II. BACKGROUND & RELATED WORK

Modern LLMs, such as GPT [2] and LLaMA [3], are based on the Transformer architecture and employ a decoder-only structure. Figure 1 shows a three-layer LLM, where nodes and edges indicate Transformer layers and dependencies between the layers, respectively. The Transformer layers are executed in the order denoted by the numbers, and the nodes that use the same set of model parameters (i.e., nodes representing the same layer) are filled with the same colour [25].

The processing of each LLM request is logically divided into a *prefill* phase and a *decoding* phase. In the prefill phase, all input tokens designated as *prompt* are processed in parallel. This phase generates the initial output token while storing the intermediate results of computed keys and values in the GPU memory, collectively referred to as the *KV cache*. For example, in Figure 1, a prompt "How to learn English?" generates the first token "Reading" and the KV cache. The decoding phase then utilises this KV cache to generate new tokens autoregressively (i.e., "Listening", "Writing", and "Speaking"), incorporating new keys and values into the KV cache.

Despite avoiding recomputation, the KV cache exacerbates the huge memory consumption of LLMs. It has therefore been an active area of research in recent years, with numerous LLM serving systems proposed to address various aspects of KV cache management. **1) KV Cache Compression.** Substantial works save the memory consumption of the KV cache via quantization and sparsity [13]–[18]. **2) Memory Management for KV Cache.** To increase GPU utilization, several works propose efficient memory management for the KV cache [20]. Kwon *et al.* propose PagedAttention [12] that allows KV cache to be stored in non-contiguous paged memory, reducing memory fragmentation. Gao *et al.* propose a hierarchical KV caching system that utilizes cost-effective storage media to store more KV caches [21]. **3) Request Migration across GPUs.** To fully utilize the compute and memory resource, several works disaggregate each request's prefill and decode phase into separate GPUs [6], [26]–[28]. However, they fix the placement of LLM requests during the memory-intensive decoding phase, even if there is a significant KV cache load imbalance. Instead, Sun *et al.* propose Llumnix [23], an LLM serving system that supports live migration for the KV cache across GPUs during the decoding phase. In other words, it introduces near-zero downtime by pipelining the computation and memory transfer. Similarly, Fu *et al.* propose ServerlessLLM [24], a serverless LLM serving system via a two-stage live migration. Moreover, Wu *et al.* co-migrate requests and adapters for a LoRA LLM serving system [29].

The first two types of work above focus on optimising the management of the KV cache within a single GPU, which is orthogonal to ours. Integrating these works can improve the memory efficiency of each GPU. The works most relevant to us are Llumnix [23] and ServerlessLLM [24], which support KV cache migration between GPUs. However, as discussed in § I and § III, several challenges need to be addressed, including unpredictable request patterns, high migration overhead, and theoretical guarantee. To overcome these challenges, our MELL develops a new multi-GPU KV cache management with an adaptive request migration mechanism for dynamic resource levels and an online KV cache scheduling algorithm that limits the number of GPUs and the number of migrations.

III. MOTIVATION

This section analyses the KV cache's characteristics, identifies its main bottleneck, and shows the potential for optimisation, which motivates the design of our MELL.

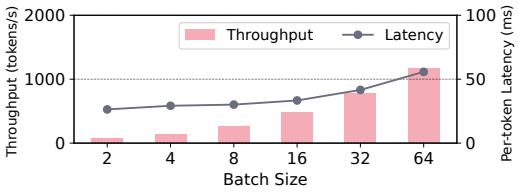
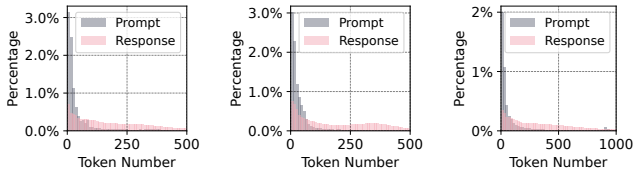


Fig. 3: Throughput and per-token decoding latency of serving LLaMA-13B in a prompt length of 100 as batch size increases.



(a) Vicuna-13B (b) Koala-13B (c) ChatGPT

Fig. 4: The distribution of prompt and response length.

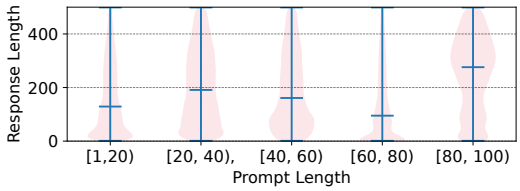


Fig. 5: The distribution of response lengths under various prompt lengths in Vicunna-13B.

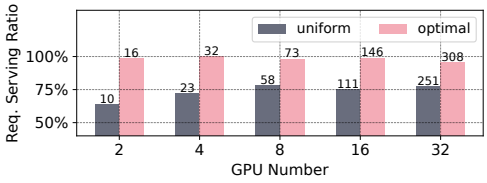


Fig. 6: The request serving ratio of an LLM serving system without request migration and with migration enabled. The number above each bar denotes the number of requests served.

Finding 1. *KV caches make existing LLM serving systems memory-bound, under-utilising GPU processing power and thus limiting serving throughput.*

Figure 2 shows the memory cost of a request fed to the LLaMA-13B model [3] on an A100 GPU with 40 GB of memory. In the prefill phase, some memory space is pre-allocated for the KV cache according to the prompt length. In the decoding phase, the size of the KV cache grows linearly with the increasing number of tokens generated. For a request with a maximum length of 4096 tokens, the memory cost of the KV cache is about 3.2 GB. After storing the model parameters of LLaMA-13B (roughly 24 GB), the A100 GPU can only support a maximum batch of 5 requests. However, as shown in Figure 3, the latency per token remains relatively stable, and the throughput continues to increase as the batch size grows from 2 to 16 when memory is not bounded (in other words, the request length is short).

An intuitive idea to improve memory utilisation is to route each incoming request to a GPU with enough memory to hold the request's KV cache. However, this is difficult to implement, as discussed below.

Finding 2. *It is difficult to predict the maximum KV cache required for an LLM request due to the inherent unpredictability of the response length generated.*

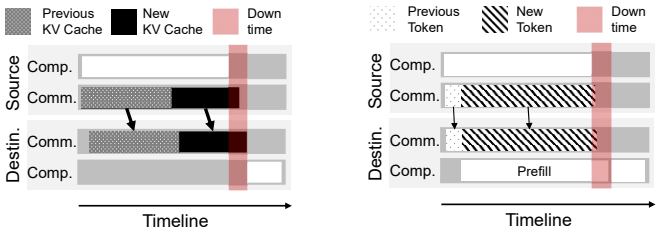
The preliminary experiment is conducted on LMSYS-Chat-1M [30] and WildChat [31], two large-scale datasets containing real chatbot conversations. Figure 4 shows the distribution of token numbers for responses in the three popular LLMs [2], [32], [33]. The distributions of token numbers vary considerably between different LLM models, with a wide range of possible values. Moreover, as shown in Figure 5, the same prompt length can yield disparate response lengths. Consequently, it is challenging to ascertain the response length based on the length of the corresponding prompt. Although a few existing works try to predict the response length [34], the prediction performance is poor (i.e., the accuracy is lower than 60% even in a length range granularity of 100 [28]).

Finding 2 highlights the challenge of scheduling algorithms without knowledge of KV cache size.

Finding 3. *Scheduling the placement of requests during their decoding phase can serve more requests simultaneously by fully utilizing multiple GPUs' memory for the KV cache.*

We evaluate the request serving ratio of an LLM serving system without request migration during the decoding phase and with migration enabled. The request serving ratio is the number of requests whose KV cache is retained in GPU memory. As shown in Figure 6, request migration allows an LLM serving system to handle $23 \sim 60\%$ more requests than a system without request migration.

Several works propose live migration for KV caches to migrate the running LLM requests without long service interruption, such as Llumnix [23] and ServerlessLLM [24]. However, their scheduling is based on a heuristic design without a performance guarantee in theory. For example, Llumnix adopts a load balancing strategy between GPUs by swapping with the lowest load and highest load repeatedly [23]. Moreover, these works only focus on how to achieve the liveness of the migration but overlook the migration overhead on computation and communication resources as follows.



(a) Llumnix (b) ServerlessLLM

Fig. 7: Two existing KV caches live migration approaches.

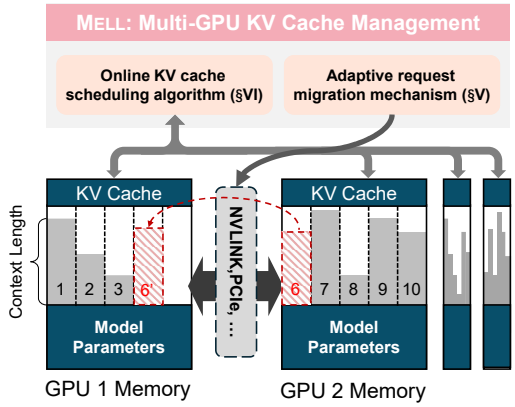


Fig. 8: System architecture of MELL.

Finding 4. *Existing migrations for the KV cache are either compute-intensive or communication-intensive.*

Figure 7 shows the workflow of the LLM request migration in Llumnix [23] and ServerlessLLM [24]. The live migration of Llumnix employs the intrinsic append-only nature of the KV cache to facilitate the concurrent transfer of the KV cache copy of the legacy tokens and the decoding computation for the new tokens. However, each migration needs a huge amount of KV cache transferred between GPUs, burdening the inter-GPU bandwidth. ServerlessLLM exhibits a comparable approach, albeit with a transformation between tokens and the KV cache, whereby the tokens are transmitted instead of the KV cache. Chunked prefill allows prefills to be batched together with decode requests. However, the latency of requests in the decoding phase will slow down by up to 2.5x when they co-execute a migrating request with a prefill requirement [28].

These findings call for a new multi-GPU KV cache management in the LLM serving system, fully utilizing GPU memory and limiting the request migration number.

IV. System Overview

This section gives an overview of MELL's design. First, we clarify three primary design goals of MELL as follows.

- **GPU cost efficiency**: Given the high cost and scarcity of GPUs in the current market, the objective of MELL is to reduce the number of GPUs required to process LLM requests in a cost-effective manner.

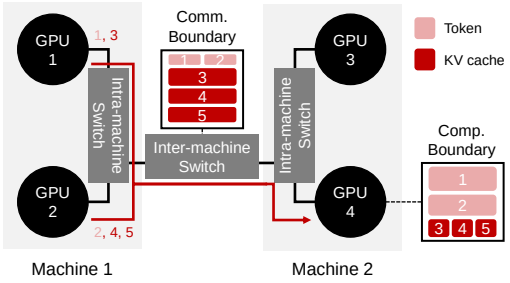


Fig. 9: An example of adaptive request migration.

- **Online strategy planning**: New requests may come at any time, and running requests may be completed. Thus, MELL needs to update the scheduling of the requests' KV caches with incomplete knowledge of the future.
- **Restricted-performance impact**: As a consequence of the request migration, the communication or computation resources are occupied and the normal running requests are influenced. Consequently, MELL mitigates the impact on the performance of the other requests.

A system overview of MELL is shown in Figure 8, where MELL is integrated into the existing multi-GPU LLM serving framework. MELL is not only a scheduling algorithm, but also a set of modules that optimize KV cache management across GPUs. It employs two key components: an adaptive request migration mechanism (refer to § V) and an online KV cache scheduler (refer to § VI). The former aims to balance the computational and communication resource overhead to minimise the negative impact caused by request migration. The latter aims to minimise the number of GPUs by migrating the LLM cache of requests across GPUs with different workloads.

The system's life cycle is composed of multiple epochs. At the beginning of each epoch, the instances send their state information (including request number and memory cost of each request's KV cache) to the cluster monitor of MELL. According to the state information, MELL generates the updated KV cache scheduling strategy via the online scheduling algorithm and sends the strategy to the instances. Then, according to the given strategy and the communication and computation capacity of the system, the instances can migrate the requests by following the adaptive request migration mechanism. For example, in Figure 8, GPU 2 migrates request 6 to GPU 1 to reduce the load on GPU 2.

V. Adaptive Request Migration

As discussed in **Finding 4** in § III, existing LLM request migrations are either compute-intensive (i.e., token transfer) or communication-intensive (i.e., KV cache transfer). This fixed migration strategy can cause resource congestion when multiple requests must be migrated simultaneously. Therefore, mitigating the negative impact of request migration by balancing its communication and computation costs becomes a critical challenge that needs to be addressed by MELL.

To address this challenge, we propose an adaptive request migration mechanism in MELL. Its main idea is to first

identify the idle computational and communication resources in the system that can be used by the request migration without affecting the normal operation of the system. Next, it migrates each request by transferring either tokens or KV cache, and orchestrates all requests to be migrated to make the consumption within the boundary. As shown in Figure 9, its workflow consists of the following steps.

Boundary Profiling. We first define the *communication boundary* of a GPU communication link as a certain amount of data that can be transferred over the link. This boundary is set to ensure that the data can be transferred in a limited amount of time. This is because the GPU memory occupied by the migrating requests cannot be released until the transfer is complete. We also define the *computation boundary* of an instance with a batch size as a certain number of tokens to be prefilled due to migration. This is because a prefill computation with long tokens interferes with the co-located computation, while one with short tokens does not since it can use idle computation resource [28]. At the beginning of the system, we identify the communication boundary for every link and the computation boundary for every instance via offline profiling. The boundary information will be shared with all instances in the system.

Hybrid Migration. Given a new KV cache scheduling strategy, each instance migrates requests according to the boundary. Each instance needs to divide its requests to be migrated into two classes. The first class includes the requests transferred as KV cache [23]. The second class includes the requests transferred as tokens and then prefilled in the destination instances [24]. The division is formulated as a two-bin-packing problem that can be solved using a greedy algorithm (e.g., first-fit or best-fit).

Global Consensus. Multiple instances can use the same link and migrate requests to the same instance. For example, in Figure 9, GPU 1 and GPU 2 use the inter-machine switch to migrate requests to GPU 4. They may exceed the boundaries if the above division is done without cooperation. To avoid this, each instance runs the algorithm considering all requests to be migrated in the system, not just its own requests. The global division is still a two-bin packing problem, where each request can choose either token transfer or KV cache transfer.

VI. Online KV Cache Scheduling

This section presents the online KV cache scheduling algorithm in MELL. The system model of a multi-GPU LLM serving system is first presented, followed by details of our algorithm and theoretical performance analysis.

A. System Model

a) GPU Cluster: We consider a set of homogeneous GPUs, denoted by J and the memory capacity for the KV cache in each GPU is denoted by C.

b) Requests: Users can send a set of LLM requests denoted by I to the system within time slots T, and the arrival time of request $i \in I$ is denoted by $a_i \in T$. These requests have different token numbers to be processed (i.e., prefilled and decoded) due to the diverse tasks of users. The memory usage of the KV cache of request $i \in I$ at time $t \in T$ is S_i^t, and it linearly increases with the number of tokens processed before the request is completed, i.e., $S_i^t \geq S_i^{t-1}$.

c) Serving Strategy: When request i arrives, it is assigned to a GPU for processing. We denote a serving strategy for T as $\mathbf{x} = \{\mathbf{x}^t\}_{t\in T}$ in which $\mathbf{x}^t = \{x_{i,j}^t\}_{i\in I, j\in J}$ is defined as

$$x_{i,j}^t = \begin{cases} 1, & \text{if request } i \text{ runs on GPU } j \text{ at time } t \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

Given a strategy $\mathbf{x}$ for T, the system should ensure that any GPU at any time slot must have enough memory space to process the allocated request, i.e.,

$$\sum_{i\in I} x_{i,j}^t S_i^t \leq C, \quad \forall j \in J, \forall t \in T. \tag{2}$$

d) System Cost: y_j^t indicates whether any requests are running on GPU j at time t and $y_j^t = \min\{1, \sum_{i\in I} x_{i,j}^t\}$. Therefore, the number of GPUs needed to serve the set of inference requests I by strategy $\mathbf{x}$ within time slots T is

$$B(\mathbf{x}) = \max_{t\in T} \sum_{j\in J} y_j^t. \tag{3}$$

B. Problem Formulation

For a serving strategy $\mathbf{x}$ to minimize the number of GPUs needed for a set of requests I, we formulate a *KV cache scheduling problem* in a multi-node multi-request system

$$\begin{aligned} \min B(\mathbf{x}) &= \min \max_{t\in T} \sum_{j\in J} y_j^t \\ \text{s.t.} \sum_{i\in I} x_{i,j}^t S_i^t &\leq C, \quad \forall j \in J, \forall t \in T. \end{aligned} \tag{6}$$

Solving the problem poses the following challenges. First, the arrival and completion of requests are unpredictable, limiting decision-making to the available information. This makes it difficult to approach the global optimum, as the system cannot anticipate all future needs. Second, the memory required for each request S_i^t is dynamic, growing over time until the request is completed. This requires constant adjustment of resource allocations, making it difficult to plan and optimise resource usage. Third, the decision at any moment resembles a bin packing problem, which is NP-complete [35]. Our problem, however, introduces greater complexity as historical choices influence each decision, complicating the resolution process significantly beyond the NP-complete framework.

C. Online Algorithm Design

According to the characteristics of LLM serving, we design an online algorithm for the KV cache scheduling problem motivated by [36]–[38]. It allocates incoming LLM requests based on GPU memory and request requirements, then updates allocations by migrating requests between GPUs to adapt to workload fluctuations. Unlike existing scheduling algorithms focusing primarily on immediate state changes, our algorithm takes a long-term view of scheduling to minimize space

★$J.Allocate(i)$:

1: ***Allocate T-request:*** For all L-GPU $j \in J$ with enough memory to fit i, allocate i to GPU j with the highest priority. Otherwise, allocate i to the most recently activated T-GPU.
2: ***Allocate S/M-request:*** For all L-GPU $j \in J$ with $S^t_{i_L} + S^t_i < C$, i_L is the L-request in j. Allocate i to j with the highest priority. Depart and re-allocate any T-request that exists in j. Otherwise, allocate i to the most recently activated S/M-GPU.
3: ***Allocate L-request:*** Activate a new GPU j, $J = J \cup \{j\}$. Allocate i to j. Move an S/M-request from an S/M-GPU j' to j if possible. Then fulfil j' with S/M-request from the most recently activated S/M-GPU.

★$J.Depart(i)$:

Assume request i is processed by GPU $j \in J$ currently.

1: ***GPU j is the most recent activated GPU:*** Remove i from j.
2: ***Depart T-request:*** If j is T-GPU, move a T-request from the most recently activated T/M-GPU to j. Otherwise, move a T-request from the most recently activated T-GPU to fit in j.
3: ***Depart S/M-request:*** If j is an S/M-GPU, move an S/M-request from the most recently activated S/M-GPU to j. Re-allocate any T-request that may exist in j. If j is an L-GPU, move an S/M-request to j from an S/M-GPU j' with the highest priority for GPU j. Then, fulfil j' with S/M-request from the most recently activated S/M-GPU.
4: ***Depart L-request:*** Depart and re-allocate all other requests in j.

★$J.Update(i)$:

Assume request i is processed by GPU $j \in J$ currently.

1: ***T/S-request → S/M-request:*** Depart i and re-allocate i.
2: ***M-request → L-request:*** If j is a L-GPU, depart i and re-allocate i. If j is an M-GPU and overload occurs after the update, depart and re-allocate all other requests in j.
3: ***L-request → L-request:*** If overload occurs after growth, depart and re-allocate all other requests in j.

Fig. 10: Three request operations for request i on GPU cluster J for multi-GPU KV cache scheduling.

Algorithm 1: Overall Workflow

Input: LLM request set I, GPU cluster J
for $t \in T$ **do**
  for $i \in \{i \mid S^t_i > 0 \vee S^{t-1}_i > 0\}$ **do**
    if request i arrives at t **then**
      $J.Allocate(i)$
    else if request i is completed at t **then**
      $J.Depart(i)$
    else if request i's type changes at t **then**
      $J.Update(i)$
  for *GPU $j \in J$ processing no request* **do**
    terminate GPU j, $J = J - \{j\}$

fragmentation and avoid creating unused fragmented spaces. It can achieve near-optimal allocation globally rather than just providing short-term solutions.

Priority-aware GPU Categories. We first classify the requests into four categories based on their KV cache sizes: L (Large), M (Medium), S (Small), and T (Tiny). Request i is an L-request if S^t_i is within $(C/2, C]$; M-request if S^t_i is within $(C/3, C/2]$; S-request if S^t_i is between $(C/4, C/3]$; T-request if S^t_i ranges from $(C/8, C/4]$. For requests smaller than $C/8$, we group them into multi-items with sizes in the range $(C/8, C/4]$. GPUs are categorised based on the largest type of request they process: GPU j is labelled as an L, M, S, or T-GPU if its largest request in category j is an L, M, S, or T-request. We also define a priority relationship from GPUs j to j', which is determined by factors including the workload (e.g., request number and idle GPU memory) of GPU j' and the distance between GPUs j to j'. For example, a GPU j' that handles fewer requests, has more GPU memory, or is on the same machine as GPU j will be assigned a higher priority for GPU j. The weights of different factors are set by the LLM service provider. Besides, we define a priority of GPU j that is only determined by the workload of GPU j for the allocation of incoming requests.

Request Allocation/Depart/Update. The arrival of new requests, the departure of completed requests, and the growth of the KV cache of running requests can all lead to GPU underloaded or overloaded. Therefore, Algorithm 1 takes the LLM request set I and GPU cluster J as input to update the allocation based on three operations: (1) allocating new requests to the most appropriate GPU, (2) dropping completed requests to free resources, and (3) updating the position of running requests along with their processing. Details of each operation are given in Figure 10, which guarantees that our algorithm maintains a near-optimal number of GPUs, as proven in § VII.

Request Operation Batching. The request operations in Figure 10 are designed to efficiently manage individual requests' allocation, departure, and update. However, overlapping operations can occur when multiple requests need to perform these operations simultaneously, resulting in redundant request migration. To address this problem, we introduce *request operation batching*. This approach combines and optimizes operations as a unified group rather than discretely, minimizing unnecessary resource allocation and migration. Implementing operation batching is critical to ensure efficient request migration within our framework, especially in high-demand scenarios. Given an operation set O, the steps for operation batching are: (1) Execute all $Depart()$ in O. Instead of executing the possible migration caused by $Depart()$, add them into an operation buffer B. (2) Execute all $Update()$ in O. Instead of executing the possible migration caused by $Update()$, add them into buffer B. Check B and remove unnecessary movement. (3) Execute all $Allocate()$ in O. Check B and remove unnecessary movement. (4) Execute all operations in the buffer.

VII. Analysis

Theorem 1. *Given any set of LLM requests I, the allocation obtained by our algorithm satisfies all the following properties (with a constant number of exceptions):*

1) *M-GPU process two M-requests, possibly one T-request.*
2) *S-GPU process three S-requests.*
3) *T-GPU memory usage is at least 75%.*
4) *L-GPU j process no S/M-request only if no M/S-request in the M/S-GPU can fit in j.*
5) *T-GPU exist only if all GPU memory utilisation of L/M-GPU is at least 75%.*

Proof. In the following, we discuss every operation separately. *1) Allocate/Depart T-request:* The preference for the L-GPU in $Allocate()$ and the attempt to replenish the L-GPU with M/S-requests in $Depart()$ ensure that property 4 is not violated. The remaining properties remain unaffected. *2) Allocate/Depart M/S-request:* The preference for L-GPU in $Allocate()$ and the attempt to refill the L-GPU with M/S-request in $Depart()$ ensure that property 4 is not violated. The remaining properties remain unaffected. *3) Allocate/Depart L-request:* Allocation/departure of L-request triggers $Allocate()/Depart()$ of other types of requests, which are shown to satisfy the properties. *4) Update Operation:* $Update()$ consist of $Allocate()$ and $Depart()$; hence, there is no violation of properties. □

Theorem 2. *For any request set I, given the scheduling algorithm A that maintains the allocation fulfils all the properties in Theorem 1, the competitive ratio of A is at most $4/3$.*

Intermediate value *weight* will be introduced for the following proofs. Each request x will have a corresponding weight $w(x)$, the total weight $W(I)$ of LLM request set I is the sum of the weights of all the requests, i.e. $W(I) = \sum_{x \in I} w(x)$. In addition, we divide the L-requests into two types: *Single* type if there is no M/S-request in this L-GPU, otherwise *Combined*. The number of *single* request is $\mathcal{S}$ and the number of *combined* request is $\mathcal{C}$. The weight of a single L-request is $w(x) = 1$; combined L-request is $w(x) = 5/6$; M-request is $w(x) = 1/2$; S-request is $w(x) = 1/3$; T-request is $w(x) = 0$, i.e. the T-requests lead to no difference in weight.

Lemma 2.1. *Given a scheduling algorithm A and request set I, the allocation is denoted as $A(I)$. If $A(I)$ fulfils all the properties in Theorem 1, we have $|A(I)| \leq W(I) + c$, where $|A(I)|$ is the number of GPUs in $A(I)$ and c is a constant.*

Proof. To prove $|A(I)| \leq W(I) + c$, we need to prove the average weight of GPUs in $A(I)$ is greater or equal to 1. By the properties in Theorem 1, we know:

- Weight of M-GPU is $1/2 + 1/2 = 1$.
- Weight of S-GPU is $1/3 + 1/3 + 1/3 = 1$.
- Weight of an L-GPU containing a single L-request is 1.

The following will prove that the average weight of L-GPUs handling M/S requests is greater than or equal to 1. By the definition of the couple L-requests, it is easy to see that at least $\lfloor \frac{\mathcal{C}}{2} \rfloor$ combined L-requests can fit with $\lfloor \frac{\mathcal{C}}{2} \rfloor$ M/S-requests. So the total weights of L-GPUs handling M/S requests are at least $\frac{\mathcal{C}}{2} * 1/3 + \frac{\mathcal{C}}{2} * 5/6 = \mathcal{C}$. So $A(I)$ is bounded by $W(I) + c$. □

Lemma 2.2. *Given request set I, $OPT(I) \leq 3/4 W(I)$ where $OPT(I)$ is the optimal allocation of I.*

Possible Combination	Weight
L,LT,LTT	$1 < 4/3$
LM	$5/6 + 1/2 = 4/3$
LS	$5/6 + 1/3 < 4/3$
MM	$1/2 + 1/2 < 4/3$
MSS	$1/2 + 1/3 + 1/3 < 4/3$
SSS	$1/3 + 1/3 + 1/3 < 4/3$

TABLE I: Weight of GPUs in each type

Proof. To prove $OPT(I) \leq 3/4 W(I)$, the weight of all the possible GPUs in $OPT(I)$ is shown in Table I □

Proof of Theorem 2. The allocation for the request list I generated by algorithm A is denoted as $A(I)$. The total size of requests $i \in I$ is $S(I) = \sum_{i \in I} S_i$. The proof will be divided into the following two separate cases:
Case 1: There is T-GPU in $A(I)$. By the fifth property, all the L-GPU and M-GPU are at least $3/4$ full. And the S-GPU is also at least $3/4$ full since the size of S-requests is in $(C/4, C/3]$ and each S-GPU processes 3 S-requests. Thus all the GPUs are at least $3/4$ full (except a constant number of *latest GPUs*). Therefore we have: $|P_A(I)| \leq \frac{4}{3} S(I) \leq \frac{4}{3} \mathrm{OPT}(I)$.
Case 2: There is no T-GPU in $A(I)$. From Lemma 2.1 and Lemma 2.2, we can conclude that the inequality $|A(I)| \leq W(I) + c \leq 4/3 \cdot OPT(I) + c$ holds. □

Theorem 3. *Given a GPU allocation $A(I)$ of requests set I, which fulfils all the properties in Lemma 2.1, the maximum number of request migrations caused by an operation is ten.*

Proof. We discuss the operations separately. Noted that allocation and departure in the latest GPU do not result in any request migration; therefore, in the following discussion, the scope of the non-latest GPU is discussed by default.

- Allocate/Depart T-request (2 migrations): It is easy to observe that no request migration will be caused by allocating T-request. When departing a T-request from GPU j. 2 migration may caused by the departure: use another T-request i_T to replenish GPU j and use another T-request to replenish the T-GPU processing i_T. So allocate/depart T-request will lead to at most 2 migrations.
- Allocate/Depart M/S-request (5 migrations): Allocate M/S-request to L-GPU may cause up to two T-requests to be removed and re-allocated (two T-requests in L-GPU). Departing an S-request from an S-GPU may cause one migration of another S-request from the latest S-GPU. Departing an M-request from M-GPU j may cause 1 migration of another M-request from the latest M-GPU to j and trigger a departure of T-request, i.e., in total 4 migrations. Departing M/S-request from L-GPU will need to find another M/S to fit in the L-GPU. Worst case will trigger a departure of M-request, which may cause at most 4 migrations, i.e., in total 5 migrations.
- Allocate/Depart L-request (5 migrations): The worst case of allocating an L-request triggers a departure of an M-request and migrates it to the new GPU, i.e. may cause up to 5 migrations. Departing an L-request may cause

two T-requests or one M/S-request to be departed and reallocated. Therefore, it may cause up to 3 migrations.

- Update (10 migrations): The update operation will trigger a departure of the original type request and an allocation of the new type request, so the worst case is departing L-request (5 migrations) and allocating L-request (5 migrations), i.e., at most 10 migrations.

□

VIII. Experiment

A. Implementation

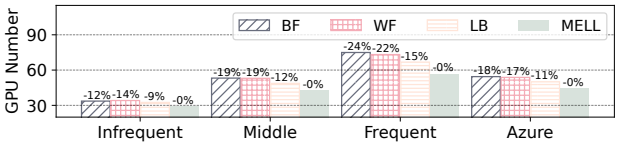
We implement a prototype of MELL on top of vLLM [39], representing the state of the art in serving systems and offering some advanced features, including paged attention and continuous batching. We deploy each instance by Ray [40] actor to implement GPU workers that execute the LLM inference and schedule the instance. The request migration of MELL is supported by the point-to-point GPU communication of tokens and KV cache in Gloo [41] similar to Llumnix [23].

B. Experimental Setup

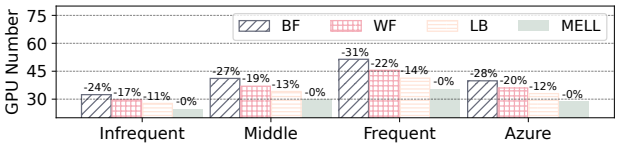
Workloads. We evaluate MELL based on LLaMA2 7B and 13B [3], one of the most popular open-sourced LLMs. The LLM request inputs are based on LMSYS-Chat-1M [30] and WildChat [31], two datasets containing more than one million real conversations collected from chatbot applications. To simulate the state-of-the-art LLMs with long-context (e.g., GPT-4o [42] and Claude 3.5 Sonnet [43]), we scale up each conversation by a factor of ten. The LLM request arrival pattern of the workload is generated from the following data. First, to simulate frequent, middle, and infrequent workloads, we use three Poisson distributions with a setting of $\lambda = 0.5$, 0.8, and 1.1, respectively. Second, we use the traces from multiple LLM inference services in Azure [6] collected on November 11th 2023, to simulate production-like workload arrival patterns and characteristics.

Node setup. We deploy MELL on a small-scale GPU cluster, including 8 NVIDIA GeForce RTX 4090 GPUs, each with 24 GB of memory, and 4 NVIDIA A100 GPUs, each with 40 GB of memory. The intra-machine GPU communication is PCIe 4.0, while the inter-machine GPU communication is 10 Gbps. We use this testbed to collect traces of request processing speeds and inter-GPU bandwidth under different workloads. We use this testbed to collect traces of request processing speeds and inter-GPU bandwidth under various workloads. We then use them to simulate the deployment of MELL on a large-scale cluster for evaluation.

Baseline. To evaluate the efficiency of MELL, we have conducted a comparative analysis with the following algorithms. (1) BF: A scheduling algorithm dispatches each incoming request to the GPU with the least but sufficient memory (i.e., Best-Fit) and does not migrate running requests between GPUs. (2) WF: A scheduling algorithm dispatches each incoming request to the least memory (i.e., Worst-Fit) and does not migrate running requests between GPUs. A similar algorithm is widely adopted by existing LLM serving systems [6], [27]. (3) LB: A scheduling algorithm dispatches incoming requests to the GPU with the least memory (i.e., worst fit) and achieves Load-Balancing via request migration by transferring the KV cache between GPUs, adopted by LLumnix [23]. These algorithms activate a new GPU if no GPU can handle an incoming request and terminate a GPU if it is idle. We implement these algorithms in our system to ensure a targeted comparison for scheduling requests.

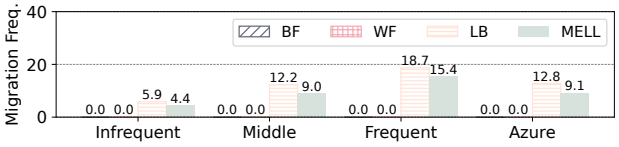


(a) LLaMA-13B on NVIDIA A100

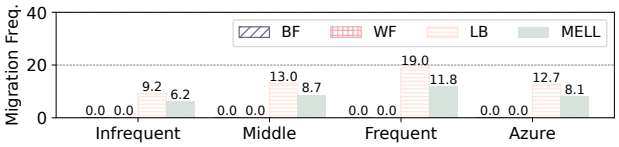


(b) LLaMA-7B on NVIDIA 4090

Fig. 11: The number of GPUs needed by different systems under the Poisson and Azure workloads. The number above the bar denotes the difference between the baseline and MELL.



(a) LLaMA-13B on NVIDIA A100



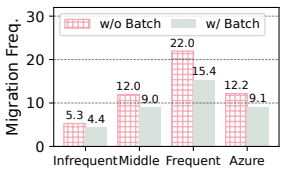
(b) LLaMA-7B on NVIDIA 4090

Fig. 12: The migration frequency in different systems under the Poisson and Azure workloads.

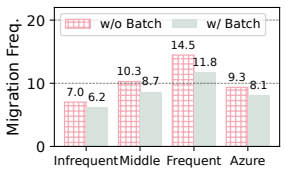
Metrics. We evaluate MELL and the baselines based on the following metrics. (1) The number of GPUs required by the LLM service provider to serve user requests. (2) The migration frequency (i.e., migrations per second) required by the LLM serving system to fully utilize idle GPU memory. (3) The GPU utilization (i.e., percentage of GPU memory in use).

C. Results

We evaluate the number of GPUs required by various LLM serving systems under different workloads, as shown in Figure 11. While the BF and WF algorithms perform similarly, the LB algorithm outperforms both because it sup-



(a) LLaMA-13B on NVIDIA V100



(b) LLaMA-7B on NVIDIA 4090

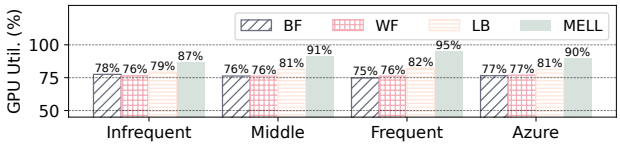
Fig. 13: The performance improvement caused by request operation batching in MELL under different workloads.

ports the request migration, balances GPU load, and improves GPU utilization. MELL combines the advantages of request migration with a design that reduces GPU fragmentation, reducing GPU demand by up to 15% compared to LB and over 20% compared to BF and WF. MELL increases its advantage as the workload's frequency grows. It is because, as the requirement of GPU grows, the existing algorithms result in more fragmented space, providing MELL greater room for optimization. Moreover, this improvement is particularly evident when using the 4090 GPU for LLaMA-7B, where the limited KV cache storage causes larger fluctuations in GPU demand, emphasizing the need for an online KV cache scheduling algorithm of MELL.

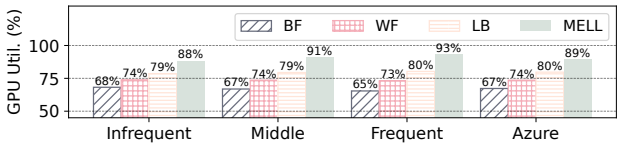
We evaluate the migration frequency in various LLM serving systems under different workloads. As illustrated in Figure 12, MELL consistently exhibits a lower migration frequency than LB, because MELL is designed with an upper limit on the number of migrations according to Theorem 3. The long-term consideration inherent to the scheduling process represents a significant advantage of MELL over LB. Additionally, the proposed operation batching can effectively reduce the incidence of unnecessary migrations. As seen in Figure 13, the technique reduces the number of migrations by up to 30% under the Poisson workloads and 25% under the Azure workload. Also, only LB and MELL support migration of running requests; BF and WF do not support migration, so their migration frequency is zero.

We evaluate the GPU utilization in various LLM serving systems under different workloads. As shown in Figure 14, MELL consistently achieves the highest GPU utilization across various workloads, with its lowest average at 88%. In comparison, the peak GPU utilization for other algorithms is around 80%, while BF's GPU utilization is as low as 65% in the Poisson workload with high arrival frequency. MELL improves the GPU utilization by $8\% \sim 28\%$ compared with the existing systems. This stark difference underscores the substantial GPU memory waste attributable to fragmented storage spaces. MELL addresses this inefficiency by effectively consolidating fragmented spaces through targeted migration requests, optimizing GPU resource utilization.

Figure 15 shows the record of GPU usage of each system under the Poisson workload. All algorithms exhibit similar performance in the initial phase. It is evident that there are

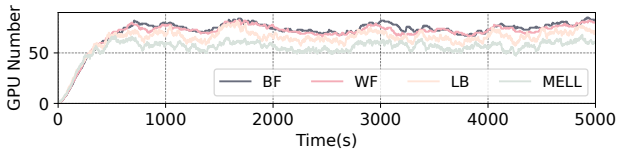


(a) LLaMA-13B on NVIDIA A100

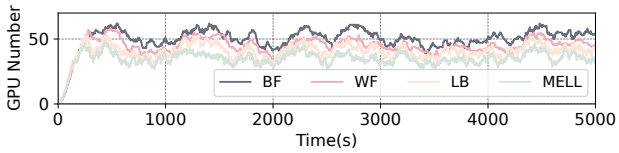


(b) LLaMA-7B on NVIDIA 4090

Fig. 14: The GPU utilization in different systems under the Poisson and Azure workloads.



(a) LLaMA-13B on NVIDIA A100



(b) LLaMA-7B on NVIDIA 4090

Fig. 15: The number of GPUs at different times in each systems under Poisson workload with high arrival frequency.

considerable fluctuations during the service phase, and the fluctuation trends are broadly similar across differing strategies. However, MELL consistently maintains the lowest GPU requirements throughout all these fluctuations.

IX. CONCLUSION

This paper proposes MELL, a memory-efficient LLM serving system via multi-GPU KV cache management. The system comprises an adaptive request migration mechanism for dynamic resource levels and an online KV cache scheduling algorithm that reduces the number of GPUs with limited request migration. We implement a prototype of MELL on LLaMA and vLLM and evaluate it based on real chatbot conversations. The results show that MELL reduces the number of GPUs by $9\% \sim 31\%$ and increases the GPU utilization by $10\% \sim 43\%$ on a Poisson simulated workload and a real workload from Azure compared to the existing LLM serving systems. In future work, we will investigate multi-GPU KV cache management for LLMs with parameter sizes that exceed the capacity of a single GPU.

REFERENCES

[1] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, in *NIPS*, 2020, pp. 1877–1901.

[2] OpenAI, "Gpt-4 technical report," 2024. [Online]. Available: https://arxiv.org/abs/2303.08774

[3] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," 2023. [Online]. Available: https://arxiv.org/abs/2302.13971

[4] Q. Hu, Z. Ye, Z. Wang, G. Wang, M. Zhang, Q. Chen, P. Sun, D. Lin, X. Wang, Y. Luo, Y. Wen, and T. Zhang, "Characterization of large language model development in the datacenter," in *NSDI*, 2024, pp. 709–729.

[5] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A distributed serving system for Transformer-Based generative models," in *OSDI*, 2022, pp. 521–538.

[6] P. Patel, E. Choukse, C. Zhang, A. Shah, I. Goiri, S. Maleki, and R. Bianchini, "Splitwise: Efficient generative llm inference using phase splitting," in *ISCA*, June 2024.

[7] Z. Hong, J. Lin, S. Guo, S. Luo, W. Chen, R. Wattenhofer, and Y. Yu, "Optimus: Warming serverless ml inference via inter-function model transformation," in *EuroSys*. Association for Computing Machinery, 2024, p. 1039–1053.

[8] J. Chen, W. Xu, Z. Hong, S. Guo, H. Wang, J. Zhang, and D. Zeng, "Otas: An elastic transformer serving system via token adaptation," in *INFOCOM*, 2024, pp. 1–10.

[9] S. Ye, J. Du, L. Zeng, W. Ou, X. Chu, Y. Lu, and X. Chen, "Galaxy: A resource-efficient collaborative edge ai system for in-situ transformer inference," in *INFOCOM*, 2024, pp. 1–10.

[10] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, J. Heek, K. Xiao, S. Agrawal, and J. Dean, "Efficiently scaling transformer inference," in *MLSys*, 2023, pp. 606–624.

[11] Y. Chen, S. Qian, H. Tang, X. Lai, Z. Liu, S. Han, and J. Jia, "LongloRA: Efficient fine-tuning of long-context large language models," in *ICLR*, 2024.

[12] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *SOSP*, 2023, p. 611–626.

[13] M. Adnan, A. Arunkumar, G. Jain, P. Nair, I. Soloveychik, and P. Kamath, "Keyformer: Kv cache reduction through key tokens selection for efficient generative inference," in *MLSys*, 2024, pp. 114–127.

[14] Z. Zhang, Y. Sheng, T. Zhou, T. Chen, L. Zheng, R. Cai, Z. Song, Y. Tian, C. Re, C. Barrett, Z. Wang, and B. Chen, "H2o: Heavy-hitter oracle for efficient generative inference of large language models," in *NIPS*, 2023.

[15] G. Xiao, Y. Tian, B. Chen, S. Han, and M. Lewis, "Efficient streaming language models with attention sinks," in *ICLR*, 2024.

[16] Z. Liu, A. Desai, F. Liao, W. Wang, V. Xie, Z. Xu, A. Kyrillidis, and A. Shrivastava, "Scissorhands: Exploiting the persistence of importance hypothesis for LLM KV cache compression at test time," in *NIPS*, 2023.

[17] Z. Liu, J. Yuan, H. Jin, S. Zhong, Z. Xu, V. Braverman, B. Chen, and X. Hu, "Kivi: A tuning-free asymmetric 2bit quantization for kv cache," *arXiv preprint arXiv:2402.02750*, 2024.

[18] S. Ge, Y. Zhang, L. Liu, M. Zhang, J. Han, and J. Gao, "Model tells you what to discard: Adaptive KV cache compression for LLMs," in *ICLR*, 2024.

[19] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Ré, I. Stoica, and C. Zhang, "Flexgen: high-throughput generative inference of large language models with a single gpu," in *ICML*, 2023.

[20] W. Lee, J. Lee, J. Seo, and J. Sim, "Infinigen: Efficient generative inference of large language models with dynamic kv cache management," in *OSDI*, 2024, pp. 155–172.

[21] B. Gao, Z. He, P. Sharma, Q. Kang, D. Jevdjic, J. Deng, X. Yang, Z. Yu, and P. Zuo, "Cost-Efficient large language model serving for multi-turn conversations with CachedAttention," in *ATC*, 2024, pp. 111–126.

[22] R. Y. Aminabadi, S. Rajbhandari, A. A. Awan, C. Li, D. Li, E. Zheng, O. Ruwase, S. Smith, M. Zhang, J. Rasley, and Y. He, "Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale," in *SC*, 2022.

[23] B. Sun, Z. Huang, H. Zhao, W. Xiao, X. Zhang, Y. Li, and W. Lin, "Llumnix: Dynamic scheduling for large language model serving," in *OSDI*, 2024, pp. 173–191.

[24] Y. Fu, L. Xue, Y. Huang, A.-O. Brabete, D. Ustiugov, Y. Patel, and L. Mai, "Serverlessllm: Low-latency serverless inference for large language models," in *OSDI*, 2024, pp. 135–153.

[25] J. Fang, Y. Yu, C. Zhao, and J. Zhou, "Turbotransformers: an efficient gpu serving system for transformer models," in *PPoPP*, 2021, p. 389–402.

[26] A. Agrawal, N. Kedia, A. Panwar, J. Mohan, N. Kwatra, B. Gulavani, A. Tumanov, and R. Ramjee, "Taming throughput-latency tradeoff in llm inference with sarathi-serve," in *OSDI*, 2024, pp. 117–134.

[27] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, and X. Jin, "Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving," in *OSDI*, 2024, pp. 193–210.

[28] C. Hu, H. Huang, L. Xu, X. Chen, J. Xu, S. Chen, H. Feng, C. Wang, S. Wang, Y. Bao, N. Sun, and Y. Shan, "Inference without interference: Disaggregate llm inference for mixed downstream workloads," 2024. [Online]. Available: https://arxiv.org/abs/2401.11181

[29] B. Wu, R. Zhu, Z. Zhang, P. Sun, X. Liu, and X. Jin, "dLoRA: Dynamically orchestrating requests and adapters for LoRA LLM serving," in *OSDI*, 2024, pp. 911–927.

[30] L. Zheng, W.-L. Chiang, Y. Sheng, T. Li, S. Zhuang, Z. Wu, Y. Zhuang, Z. Li, Z. Lin, E. Xing, J. E. Gonzalez, I. Stoica, and H. Zhang, "LMSYS-chat-1m: A large-scale real-world LLM conversation dataset," in *ICLR*, 2024.

[31] W. Zhao, X. Ren, J. Hessel, C. Cardie, Y. Choi, and Y. Deng, "Wildchat: 1m chatGPT interaction logs in the wild," in *ICLR*, 2024.

[32] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. Xing, H. Zhang, J. E. Gonzalez, and I. Stoica, "Judging LLM-as-a-judge with MT-bench and chatbot arena," in *NIPS Datasets and Benchmarks Track*, 2023.

[33] X. Geng, A. Gudibande, H. Liu, E. Wallace, P. Abbeel, S. Levine, and D. Song, "Koala: A dialogue model for academic research," Blog post, April 2023. [Online]. Available: https://bair.berkeley.edu/blog/2023/04/03/koala/

[34] Z. Zheng, X. Ren, F. Xue, Y. Luo, X. Jiang, and Y. You, "Response length perception and sequence scheduling: An LLM-empowered LLM inference pipeline," in *NIPS*, 2023.

[35] R. M. Karp, *Reducibility among Combinatorial Problems*. Springer US, 1972, pp. 85–103. [Online]. Available: https://doi.org/10.1007/978-1-4684-2001-2_9

[36] W. Song, Z. Xiao, Q. Chen, and H. Luo, "Adaptive resource provisioning for the cloud using online bin packing," *IEEE Transactions on Computers*, vol. 63, no. 11, pp. 2647–2660, 2014.

[37] S. Kamali and A. López-Ortiz, "Efficient online strategies for renting servers in the cloud," in *SOFSEM*, 2015, pp. 277–288.

[38] D. Meisner, B. T. Gold, and T. F. Wenisch, "Powernap: eliminating server idle power," *SIGARCH Comput. Archit. News*, vol. 37, no. 1, p. 205–216, mar 2009. [Online]. Available: https://doi.org/10.1145/2528521.1508269

[39] vLLM, "Easy, fast, and cheap llm serving for everyone," 2024. [Online]. Available: https://github.com/vllm-project/vllm

[40] Ray, "Ray: a unified framework for scaling ai and python applications," 2024. [Online]. Available: https://github.com/ray-project/ray

[41] Meta, "Gloo: Collective communications library with various primitives for multi-machine training," 2024. [Online]. Available: https://github.com/facebookincubator/gloo

[42] OpenAI, "Openai platform document," 2024. [Online]. Available: https://platform.openai.com/docs/models

[43] Anthropic, "Anthropic platform document," 2024. [Online]. Available: https://docs.anthropic.com/en/docs/about-claude/models