FastSwitch: Optimizing Context Switching Efficiency in Fairness-aware Large Language Model Serving

Ao Shen [1 2] **Zhiyao Li** [* 3] **Mingyu Gao** [2 3]

Abstract

The rapid progress of Large Language Models (LLMs) has driven the need for efficient inference systems capable of serving numerous users and tasks concurrently. To better meet these demands, ensuring fairness in service is crucial. Preemption-based scheduling achieves fairness by dynamically adjusting request priorities. However, in the state-of-the-art inference system vLLM, the overhead caused by preemption-induced context switching remains unresolved. This inefficiency arises from three main factors: inadequate I/O utilization, GPU idle, and unnecessary I/O transmission, all of which increase latency and reduce efficiency. We introduce FastSwitch, a fairness-aware serving system that mitigates the overhead of frequent context switching in LLM serving while preserving fairness. Our approach includes innovations such as an I/O-aware KV cache manager, a Multithreading Swap Manager for asynchronous swapping handling, and a KV cache reuse mechanism to reduce the volume of data swapping. Our evaluation shows that FastSwitch achieves a speedup of 1.4-5.8$\times$ across different tail Time-to-First-Token (TTFT), with improvements of up to 5.8$\times$, 4.3$\times$, and 3.7$\times$ at P95, P99, and P99.9, respectively. Additionally, we observe the P99.9 Time-Between-Tokens (TBT) improvement of up to 11.2$\times$ and the throughput increase of up to 1.44$\times$. These results demonstrate FastSwitch's effectiveness in ensuring fairness without compromising performance, even with frequent priority adjustments.

1 Introduction

Large Language Models (LLMs) like GPT-3 (Brown, 2020), LLaMA (Touvron et al., 2023), and Qwen (Yang et al., 2024) have revolutionized AI by powering applications such as language translation, conversational agents, and code generation (Nijkamp et al., 2023; Liu et al., 2024b; Zhu et al., 2024; Hendrycks et al., 2020; Minaee et al., 2024; Gong et al., 2018; Bisk et al., 2020; OpenAI, 2024). With extensive parameters and diverse pretraining datasets, these models set new NLP benchmarks. Consequently, Model-as-a-Service (MaaS) platforms for deploying LLMs have seen widespread adoption (Zheng et al., 2023; Kwon et al., 2023; Sheng et al., 2023; Agrawal et al., 2024; 2023; Aminabadi et al., 2022). However, to efficiently process numerous inference requests, transformer-based LLM serving typically requires storing the request's key-value (KV) cache in GPU memory. While GPUs offer High-Bandwidth Memory (HBM) (Larimi et al., 2020), which significantly boosts I/O performance, this memory is both expensive and limited in capacity. The associated costs and limited availability of HBM pose challenges to scalability. Moreover, advanced features such as Chain-of-Thought (CoT) reasoning and multimodal inputs (Wei et al., 2022; Koh et al., 2024) necessitate handling longer context lengths, further increasing the demand for resources. To better serve a wide range of requests and users, while ensuring that as many users' needs are met as possible, fairness in serving becomes a critical concern. Many LLM serving systems emphasize fairness by dynamically adjusting request priorities during runtime based on Service Level Objective (SLO) metrics. Given the ever-present constraints on GPU memory, preemption mechanisms are essential for managing resources efficiently and maintaining fairness. Recent works such as VTC (Sheng et al., 2024), Andes (Liu et al., 2024a), QLM (Patke et al., 2024), FastServe (Wu et al., 2023), and LLMS (Yin et al., 2024) have explored various strategies for preemptive scheduling to ensure fairness and mitigate issues like head-of-line blocking in LLM serving systems. Multi-turn conversations present another important scenario for preemption, as the next turn in a conversation might be requested after a certain delay. In such cases, the KV cache of a completed conversation needs to be preempted and transferred to CPU memory for future use.

However, since most systems (Kwon et al., 2023; Zheng et al., 2023) adopt swapping as the default preemption approach, each preemption results in context switching. In

[*]Equal contribution [1]Department of Computer and Information Technology, Purdue University, West Lafayette, USA [2]Shanghai Qi Zhi Institute, Shanghai, China [3]Institute for Interdisciplinary Information Sciences, Tsinghua University, Beijing, China. Correspondence to: Ao Shen <shen634@purdue.edu>.

our work, context switching refers to the swapping of the KV cache of the preempted requests, requiring large KV cache to be swapped between GPU and CPU memory. In the state-of-the-art serving system vLLM, the page-based KV cache management strategy used to achieve higher throughput causes fragmented memory allocation, leading to poor utilization of PCIe I/O bandwidth. Additionally, the current scheduling design causes preemption to stall inference, leaving the GPU idling. What's more, when serving multi-turn conversations, there is redundant swap out volume of previous conversation across multiple turns. These challenges introduce significant overhead and degrade key performance metrics such as TTFT and TBT, ultimately reducing the overall Quality of Experience (QoE) (Liu et al., 2024a). As shown in Figure 1, the stall time caused by preemption can be several times longer than the inference time of a single iteration.

Addressing the overhead from frequent context switching is crucial for improving the performance and scalability of LLM serving systems. Prior works (Sun et al., 2024; Gao et al., 2024; Wu et al., 2023) propose strategies to mitigate this overhead, but they either disrupt vLLM's KV cache management, struggle with issues from asynchronous methods aimed at avoiding GPU idling, or fail to effectively handle redundant KV cache transfers. Consequently, they fail to provide a lightweight and effective solution. Intuitively, we can address this problem by reducing call stack overhead, improving bandwidth utilization, and minimizing redundant transfers. In response, we introduce FastSwitch, a new serving system that optimizes preemptive context switching in LLM inference with minimal additional cost. FastSwitch leverages an I/O-aware KV cache management strategy, the Dynamic Block Group Manager, which enhances bandwidth utilization and reduces kernel dispatch overhead by allocating memory for KV cache at a coarser granularity. Additionally, FastSwitch integrates a Multithreading Swap Manager to asynchronously manage KV cache transfers, minimizing delays from cache dependencies during context switching and improving token generation efficiency. Finally, the KV Cache Reuse Mechanism, integrated into the Dynamic Block Group Manager, manages and reuses KV cache copies across multi-turn dialogues, thereby reducing unnecessary I/O usage. As a result, FastSwitch effectively addresses the overhead from frequent context switching caused by high-frequency priority changes, ensuring fairness and responsiveness in dynamic, high-demand environments while maintaining efficient resource utilization. In summary, this paper makes the following key contributions:

- Priority-based Serving System: We propose a new priority-based serving system, FastSwitch, that enables efficient preemptive context switching in LLM serving by balancing fairness, throughput, and performance. FastSwitch synergistically integrates three optimizations to address key challenges like bandwidth underutilization, GPU idling, and unnecessary I/O usage.
- In-depth Evaluation: Demonstrate substantial improvements on NVIDIA GPUs compared with vLLM, particularly under high priority-update frequency. Specifically, we achieve a speedup in TTFT of up to 1.4-5.8×, in TBT of up to 11.2×, and an increase in Throughput of up to 1.44×.
- Ensuring Fairness without Compromising Performance: Ensure fairness among concurrent requests without compromising performance, allowing reliable and efficient LLM deployment in resource-constrained environments.

2 Background and Motivation

2.1 Preemption-based Scheduling in LLM Inference

Preemption-based scheduling is crucial in LLM inference to ensure fairness and manage requests with varying priorities due to constrained HBM. It allows high-priority tasks to preempt lower-priority ones and reallocate resources dynamically. There are two main preemption methods: recomputation, which halts a task and recomputes its KV cache upon resumption, increasing latency and resource use, especially for long contexts; and swapping, which transfers the KV cache between GPU and CPU memory, avoiding recomputation. Systems like vLLM effectively use swapping. Recent works explore various preemptive scheduling strategies: VTC (Sheng et al., 2024) addresses fair token-level scheduling, Andes (Liu et al., 2024a), FASTSERVE (Wu et al., 2023), and QLM (Patke et al., 2024) focus on mitigating head-of-line blocking and context switching latency. And LLMS (Yin et al., 2024) manages multiple LLM contexts across apps. Our design builds on vLLM by optimizing its swapping efficiency.

2.2 Motivation

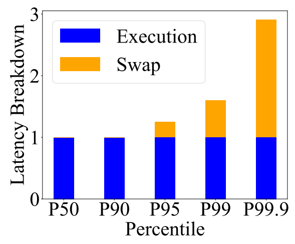

Figure 1. Latency breakdown across percentiles.

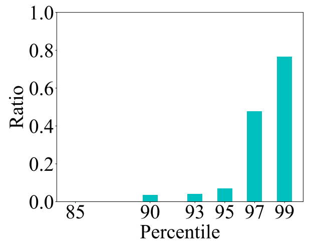

Figure 2. Only a small ratio of requests need to wait for the KV cache in most iterations.

Observation: Context Switching Overhead. Preemptive context switching, necessary for managing dynamic workloads and prioritizing tasks, introduces significant overheads, particularly from KV cache I/O operations. These affect performance metrics like Time-to-First-Token (TTFT) and Time-Between-Tokens (TBT). The impact worsens with longer contexts and increased preemption.

An experiment using the LLaMA-8B model served with vLLM on an A10 GPU involved processing 1,000 multi-turn requests from the ShareGPT dataset with request rate of 1 req/s and priority updates every 100 iterations. We normalized the latency by setting the execution time of inference to 1. The latency penalty from preemption was measured as KV cache swapping time.

Figure 1 shows that P99 latency is approximately 1.6 times higher than the P50, with swapping-induced stall time accounting for about 59.9% of P99 latency. This reveals significant performance degradation in high-stress scenarios, which becomes even more pronounced at P99.9 where the total latency increases to nearly 2x the inference time.

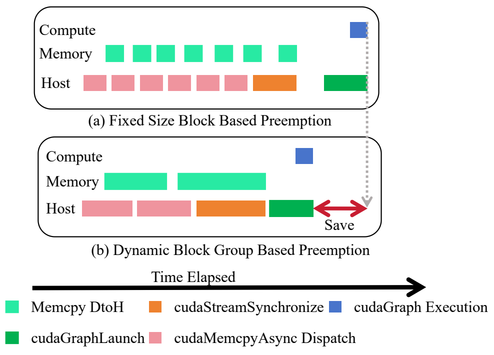

Figure 3. Timeline comparison of fixed-size block based preemption and dynamic block group based preemption.

Challenge #1: Low Bandwidth Utilization. While vLLM's KV cache management policy effectively reduces internal fragmentation by using non-contiguous virtual memory for the KV cache, optimizing swapping of the KV cache remains a significant challenge. As illustrated in Figure 3(a), performance bottlenecks arise due to sub-optimal KV cache swapping granularity, such as small 128 KB KV cache swapping granularity in LLaMA-8B. The dispatch overhead for each `cudaMemcpyAsync` call exceeds its 10 µs execution time, leading to I/O idling This issue is further exacerbated by the fact that the transfer size is below PCIe 4.0's optimal 320 KB, thus reducing efficiency. In this experimental setting, dispatch time accounts for 90%-95% of the total transmission time.

Simply increasing the block size in vLLM could cause internal fragmentation. And it would undermine the memory utilization efficiency that vLLM strives to achieve. To address this, vLLM sets the default block size to 16 tokens, aiming to strike a balance between memory efficiency and performance. Traditional approaches that preallocate KV cache memory increase fragmentation, while dynamic allocation introduces complexitiy and overhead during context switching. Therefore, developing lightweight, adaptive memory management strategies that align with vLLM's policies is critical for maintaining both efficient memory usage and larger transfer granularity to better utilize bandwidth. This remains a key challenge that needs to be addressed.

Solution 1. Instead of managing individual blocks, handling memory in larger granularities helps us maintain memory continuity and reduces the overhead associated with context switching. The key insight is to build upon the existing vLLM policy and allocate multiple contiguous blocks and encapsulating them into a higher-level abstraction. This abstraction allows for dynamic merging and splitting of block groups. This approach not only enables more efficient utilization of PCIe bandwidth during KV cache transfers but also preserves the original high throughput.

Challenge #2: GPU Idling During Preemption. We evaluated LLaMA-8B on an NVIDIA A10 GPU using a Markov priority pattern (priority-update frequency = 0.02) with 500 multi-turn conversations from ShareGPT. As shown in the figure, the impact of global priority updates across requests is most pronounced in tail cases, where a significant proportion of requests experience delays. In most scenarios, KV cache transfers between CPU and GPU memory affect only a subset of requests. Although most requests proceed without interruption, the overhead from preemption can exceed a single inference step. This leads to inference stalls, resulting in GPU remains idle.

Solution 2. By handling transfers asynchronously, we can overlap swapping and inference, reducing GPU idle time and improving token generation efficiency. This approach reduces context switching latency, prioritizes high-importance tasks, and minimizes the performance impact of swapping-related delays. However, in the scenario of LLM serving, there are several challenges to achieving comprehensive optimization and we will discuss them later.

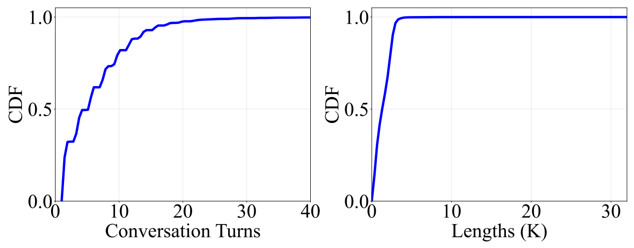

Figure 4. ShareGPT conversation turns & lengths distribution.

Challenge #3: Contaminated CPU KV Cache Copies in Multi-turn Conversations. Multi-turn conversations are prevalent in real-world LLM applications, especially in interactive systems like chatbots. Datasets such as ShareGPT (ShareGPT, 2024), which contains 100K conversations, reveal that 78% of interactions consist of multiple turns, with an average of 5.5 turns per conversation (Figure 4). These conversations often involve lengthy contexts, resulting in a large KV cache that must be maintained to preserve contextual coherence across turns. However, the repetitive nature of multi-turn conversations leads to a significant portion of the KV cache being recomputed unnecessarily in subsequent turns. This introduces redundant computation. To address this issue, AttentionStore (Gao et al., 2024) explored prefix reuse in multi-turn conversations, primarily targeting the swap in process during the prefill phase. Their approach transfers the KV cache to multi-tier storage after each conversation is completed, maintaining a full backup for future reuse. When a new conversation turn finishes, only the incremental KV cache corresponding to the new turn is swapped out, as the context from previous turns is already preserved in the multi-tier storage. This reduces the overall swap out volume. However, in the hardware setting where memory resources for KV cache are solely allocated to GPU and CPU memory, such an approach becomes challenging. Not all requests can fully utilize KV cache copies stored in CPU memory given that CPU memory is not unlimited. Intuitively, when a high-priority request requires memory allocation, the system prioritizes reclaiming memory from lower-priority requests, which invalidates lower-priority requests' KV cache backups. This presents a new challenge: to create a complete copy for prefill the next turn with prefix, how we can minimize the unnecessary removal of the previous context when the KV cache memory is preempted by other tasks.

Solution 3. Given that KV cache backups in CPU memory can be removed or contaminated by higher-priority requests, we can track the released free block groups for the KV cache copies. By continuously monitoring the validity of each block group in real-time, we identify the valid portions of the KV cache and minimize redundant KV cache swapping. Specifically, the system synchronizes the status of block groups and reuses valid smaller block group split from original block group during multi-turn conversations. Compared with completely swapping KV cache of all previous conversations in vLLM, this approach significantly reduces the volume of KV cache that might be repeatedly swapped out to CPU memory, thereby decreasing unnecessary I/O operations in constrained environment.

3 Design of FastSwitch

As shown in Figure 5, to achieve better fairness in LLM serving system, FastSwitch enhances the efficiency of preemptive context switching. To address the challenges outlined in Section 2.2, FastSwitch comprises three key optimizations that work in concert to optimize performance and resource utilization. The **Dynamic Block Group Manager** maximizes PCIe bandwidth utilization through larger-granularity memory for KV cache, effectively addressing the challenges of dynamic KV cache allocation and call stack overhead. This manager also tracks block group usage and integrates the **KV Cache Reuse Mechanism** to minimize the volume of data transferred during preemption. Building on this foundation, the **Multithreading Swap Manager** leverages async swapping to improve token generation throughput. Finally, the **Priority Scheduler** schedules high-priority requests into the running batch based on the latest priorities.

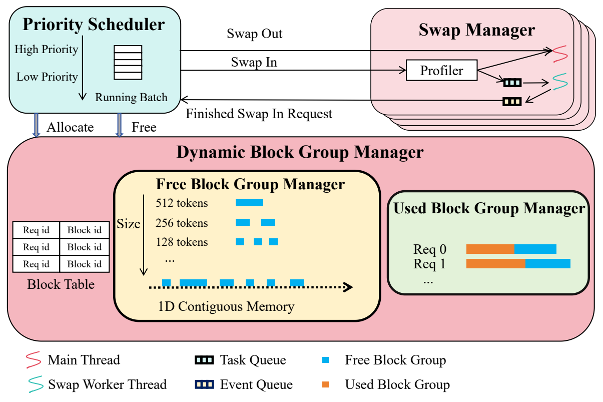

Figure 5. FastSwitch system overview.

3.1 Dynamic Block Group Manager for Increased Granularity and I/O Bandwidth Utilization

In this section, we propose the Dynamic Block Group Manager to address **Challenge #1**. Previous work such as Llumnix (Sun et al., 2024) tackled bandwidth inefficiencies caused by insufficient granularity in I/O management. They attempted to increase I/O utilization by adjusting the buffer size, but their approach was still unable to fully exploit the available bandwidth because of limited granularity. Moreover, their method introduced additional design complexity and overhead due to the necessity of a second transfer when merging data into the buffer. We introduce the Dynamic Block Group Manager, an I/O-aware KV cache allocator. This manager increases the granularity and size of KV cache allocation, reducing dispatch time and improving I/O bandwidth utilization. By leveraging the principles of buddy allocation, the Dynamic Block Group Manager aims to allocate memory blocks that closely match the size required by each request, thereby optimizing transfer efficiency. The Dynamic Block Group Manager is designed to be pluggable into existing systems. It seamlessly integrates with vLLM's KV cache management policy, maintaining high throughput while improving I/O utilization during preemption.

Dynamic Block Group Allocation and Management. The key idea behind Dynamic Block Group Manager, is analogous to the buddy allocator (Von Puttkamer, 1975) used in operating systems. The memory for KV cache is allocated in larger chunks referred to as block groups, each comprising multiple contiguous vLLM blocks. Each request is assigned one or more block groups to store its KV cache. The most recently allocated block group for a request is considered active. This active block group not only contains new KV cache for the current request but can also be split into one or more smaller block groups to serve other requests. We first perform block/page-based preallocation similar to vLLM. And then merge all blocks into an initial free block group. Based on need on both size and address, this free block group is subsequently split into two or three smaller groups. The Dynamic Block Group Manager organizes block groups through two primary subcomponents: the Free Block Group Manager and the Used Block Group Manager.

To maintain optimal memory usage, the manager supports dynamic splitting and merging of block groups. When no matching free block group is available and a request's memory requirement does not fully utilize a block group in the Used Block Group Manager, the manager can split the block group into smaller block groups. The unused portions are then reallocated to accommodate other requests as needed. Conversely, if multiple adjacent free block groups are available, they can be merged to form larger block groups. This merging enhances memory continuity and further reduces fragmentation. We initially set the expected size of the first block group for each request to 60 blocks, which corresponds to approximately 1,000 tokens when the block size is 16 tokens. The manager dynamically adjusts this size to meet the expected KV cache requirement of each request, taking into account the current availability of free KV cache.

Bandwidth Utilization Improvement. The Dynamic Block Group Manager enables larger granularity transfers by managing memory at the block group level, reducing the number of transfers and eliminating associated latency. As shown in Figure 3 (b), compared to fixed-size blocks, our design consolidates smaller memory operations into fewer, larger transfers, thereby reducing dispatch overhead and improving PCIe utilization. Llumnix introduces an additional 2-block buffer to merge the KV cache before performing a secondary transfer. However, this granularity is insufficient to fully utilize I/O bandwidth and the extra transfer brings complexity and additional latency. Moreover, simply increasing the buffer size risks excessive space usage, which contradicts vLLM's goal of minimizing waste through on-demand allocation. Our approach not only improves I/O utilization by providing better granularity but also avoids the need for secondary transfer. For example, when deploying LLaMa-8B on an NVIDIA A10 GPU, our method achieves an average granularity of approximately 20 blocks per block group. This result is observed across seven distinct frequencies, with a median priority-update frequency of 0.02(once every 50 iterations). This maximizes bandwidth efficiency while avoiding the complication associated with secondary transfer.

3.2 Multithreading Swap Manager for Optimizing Token Generation Efficiency

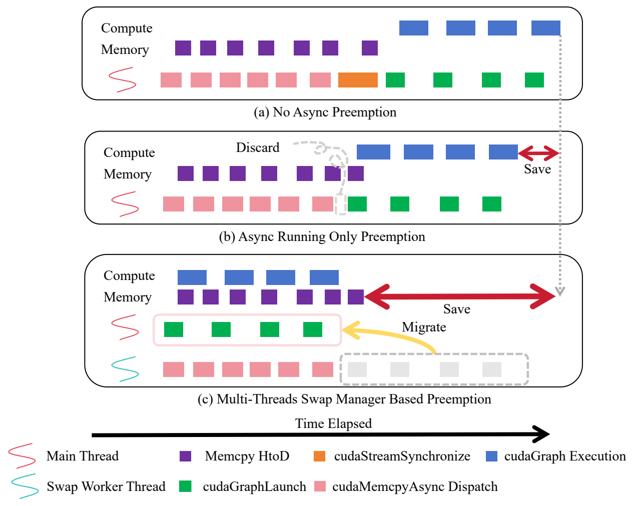

Figure 6. Comparison of varying degrees of asynchronous preemption.

To address **Challenge #2**, previous work like AttentionStore (Gao et al., 2024) supports layer-wise asynchronous swapping. However, this approach can interfere with the execution of CUDA graphs during inference and increase inference time, especially when swapping latency exceeds inference time. The core challenge lies in the nature of CUDA graphs, which are generated through static compilation. If one attempts to integrate swapping subgraphs into an existing execution graph, given that batch size is a parameter of the node in graph, it becomes necessary to iterate over all possible batch size values across both the swapping subgraph and the execution subgraph. This requires a significant amount of GPU memory for the storage of CUDA graphs because each CUDA graph needs to reserved space for input, activations, and output. Also, profiling a new graph on-the-fly every time is impractical. On the other hand, if integration is not performed, the static execution graph cannot dynamically adjust its size to accommodate temporary changes. For these reasons, the latest version of vLLM has deprecated this mechanism. Moreover, without an I/O-aware KV cache allocator to increase transfer granularity, the bottleneck of swapping lies in the overhead of the `cudaMemcpyAsync` dispatch stage rather than the execution stage as mentioned in **Challenge #1**. This issue cannot be resolved by layer-wise swapping.

FastServe (Wu et al., 2023) adopts iteration-wise transmission to predict and pre-load KV cache while supporting inference CUDA graph execution for GPU efficiency. However, predicting suitable requests for preemptive swap out can be challenging. Additionally, it does not achieve asynchronous dispatch between swapping APIs and inference kernels, thus failing to resolve the major part of overhead as stated in **Challenge #1**. It also overlooks the difficulty of maintaining a coherent order of swapping dispatches. We will discuss it in this section.

Built upon the Dynamic Block Group Manager from the first section, we now introduce the Multithreading Swap Manager to comprehensively address **Challenge #2** in an iteration-wise manner.

Adaptive Swapping Strategy. The Swap Manager employs a dynamic swapping strategy based on the system's current state. To enable informed decision-making, a profiler monitors key metrics such as the number and size of ongoing swapping operations over a recent time window. We observe that asynchronous handling of preemption is not always the optimal solution. While asynchronous swapping in generally reduces idle time and improves efficiency, it's not always the best approach. Specifically, when the total number of requests is high, but each request is relatively short, asynchronous swap in may degrade token generation efficiency. In such cases, it is more beneficial to perform synchronous swap in, as the overhead of swapping is minimal compared to the potential gain from processing a larger number of tokens. This adaptive strategy carefully balances the swapping overhead with token generation efficiency, dynamically switching between asynchronous and synchronous swap in based on workload run time metrics.

Overcoming Python GIL Limitation and Ensuring Conflict-free Dispatch Order of Multi-stream CUDA Runtime APIs. We observed that Python-based call stacks in many serving systems introduce the Global Interpreter Lock (GIL), which bottlenecks parallel execution of asynchronous tasks by limiting CPU-side kernel dispatch or API dispatch. While the execution stage remains unaffected, the restricted dispatch reduces the benefits of asynchronous optimizations and constrains overall system throughput and scalability. To mitigate this, we offloaded the dispatch process to C++, where we created a thread pool and used worker threads to dispatch the APIs and create CUDA events, thereby gaining fine-grained control over the entire process.

We also identified another implementation challenge: In the same CUDA context, the dispatch order of CUDA memcpy APIs across multiple streams needs to manage in a proper way. The key insight here is that in the execution stream, `cudaMemcpyAsync` calls will also be issued. If the swapping stream has already dispatched a large number of `cudaMemcpyAsync` operations, even if the inference stream has higher priority, it cannot preempt the execution stage of `cudaMemcpyAsync` in the swapping stream. This results in inference stalls and GPU remains idle because the `cudaMemcpyAsync` in the inference stream must wait for I/O resource to become available in order to complete. To address this issue, we implemented fine-grained synchronization control. After a certain number of dispatches, we perform synchronization to ensure that high-priority APIs can be inserted into the dispatch queue and dispatched successfully. Although this introduces a small synchronization overhead, it is insignificant compared to the performance gains achieved through overlap.

KV Cache Conflict Resolution in Asynchronous Swapping. While asynchronous KV cache transfers enable the overlap of swapping and inference operations, boosting token generation efficiency, they also introduce the risk of KV cache conflicts between the KV cache of ongoing swapping requests and newly allocated KV cache from running requests. To address this issue, we leverage the Dynamic Block Group Manager to monitor block group allocation and usage. When KV cache conflict is detected, the Swap Manager synchronizes KV cache transfer event in a fine-grained manner, minimizing resource contention and resolving KV cache conflicts. This ensures efficient operation during overlapping swapping and inference.

GPU Utilization Improvement. By allowing certain swapping operations to occur in parallel with active inference processes, the Swap Manager reduces idle time and enhances overall throughput. This asynchronous handling ensures that the majority of requests can proceed without being delayed by swapping dependencies. As shown in Figure 6, in (a), no asynchronous preemption is applied, meaning all operations, such as memory copies (`Memcpy HtoD`), `cudaGraphLaunch`, and `cudaGraph` execution, are performed sequentially. This leads to inefficient resource utilization, as tasks are serialized, resulting in longer overall inference time.

In (b), only the execution stage of `cudaMemcpyAsync` is asynchronous, while the `cudaMemcpyAsync` dispatch remains synchronous. This limits the efficiency gains as the dispatch phase still causes delays and prevents full concurrency.

In contrast, (c) implements a fully asynchronous approach, where both the dispatch and execution stage of `cudaMemcpyAsync` are asynchronous to the inference. This allows for improved concurrency and more efficient resource utilization, leading to better overall performance.

Algorithm Explanation. Algorithm 1 details the operational workflow of the Multithreading Swap Manager. The process starts by monitoring ongoing swap in operations.

Algorithm 1 Multithreading Swap Manager Algorithm

```
Initialization:
  running ← InitializeQueue()
  swapped ← InitializeQueue()
  ongoing_swap_in ← InitializeQueue()
# Recent Swapping Information
  r_info ← InitializeQueue()
for each iteration do
  # Step 1: Verify swap In Completion
  for each req in ongoing_swap_in do
    if IsCompleted(req) then
      Move(req, ongoing_swap_in, running)
    end if
  end for
  # Step 2: Handle swap In Requests
  if HasSwapIn(swapped) then
    ExecuteSwapIn(swapped)
    UpdateQueue(r_info, swapped, "SwapIn")
  end if
  # Step 3: Handle swap Out Requests
  if HasSwapOut(running) then
    ExecuteSwapOut(running)
    UpdateQueue(r_info, running, "SwapOut")
    # Step 3.1: Conflict Detection
    if DetectConflict(running, swapped) then
      Synchronize(running, swapped)
    end if
  end if
  # Step 4: Dynamic Swapping (Async Or Sync)
  decision ← Strategy(running, swapped, r_info)
  if decision == "yes" then
    MovePending(running, ongoing_swap_in)
  else
    SwapInStreamSynchronize()
  end if
end for
```

Once an asynchronous swap in is completed, the corresponding request is moved from the ongoing_swap_in queue to the running queue, and the r_info queue is updated accordingly. The manager then proceeds to handle both swap in and swap out requests. After completing swap out operations, it checks for any conflicts between the recently swap out requests and the ongoing swap in requests. If a conflict is detected, the manager performs synchronization to resolve the issue. Lastly, the algorithm employs a dynamic execution strategy, leveraging real-time profiling to optimize decision-making at each iteration. It evaluates recent swapping metrics from the recent_swap_info queue to determine whether to asynchronously proceed the swap in or to initiate a synchronization of the swap in stream to synchronize it.

3.3 KV Cache Reuse Mechanism for Efficient Multi-turn Conversations

In the last optimization of our design, to tackle **Challenge #3**, we introduce the KV Cache Reuse Mechanism to reuse the KV cache copies in CPU and handle partial cache contamination, where the KV cache in CPU memory is contaminated by higher-priority requests. This mechanism enables the reuse of partially valid KV cache, minimizing preemption overhead by reducing the volume in resource-constrained scenarios.

KV Cache Reuse Mechanism. Our mechanism focuses on retaining and efficiently managing KV cache by keeping a copy of the KV cache in CPU memory. To solve the **Challenge #3**, we designed an algorithm that effectively tracks the status of KV cache. To ensure the validity of reused KV cache, we implement a block-group-based tracking system that monitors which segments of the KV cache have been contaminated by higher-priority requests. So we can identified uncontaminated block groups that are eligible for reuse, preventing erroneous data access. As shown in Figure 7, during preemption, the KV Cache Reuse Mechanism selectively swaps out only the necessary portions of the KV cache, minimizing the KV cache swapping between CPU and GPU memory and reducing preemption latency. Furthermore, the system preallocates additional memory space for the next turn's swap out increment, which is adjacent to KV cache copy already stored in CPU memory. This proactive allocation improves memory continuity, prevents fragmentation, and ensures smoother transitions between turns. In summary, compared to vLLM this mechanism not only reduces the amount of data that needs to be swapped out during preemption, but also successfully reuses the partially valid KV cache in the constrained scenarious, eliminates the need for recomputing KV cache for repeated tokens, and significantly reduces the prefilling time—the latency involved in generating the first token of a new turn.

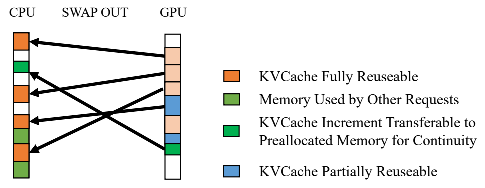
CPU
SWAP OUT
GPU
KVCache Fully Reuseable
Memory Used by Other Requests
KVCache Increment Transferable to Preallocated Memory for Continuity
KVCache Partially Reuseable

Figure 7. Workflow of the KV Cache Reuse Mechanism.

4 METHODOLOGY

We evaluate FastSwitch using the LLaMA-8B and Qwen-32B models on NVIDIA A10 and A100 GPUs. The setup includes 60 GB of CPU swap space per GPU to enable efficient context switching. Leveraging PCIe 4.0 with a x16 interface, each GPU achieves a theoretical bandwidth of 32 GB/s per direction (64 GB/s bidirectional).

4.1 Workloads

We utilize the Multi-Round ShareGPT (ShareGPT, 2024) dataset to simulate extended, realistic conversations, as depicted in Figure 4. To preserve authenticity, we retain the original input and output lengths. From the dataset, we randomly select 1,000 multi-turn conversations and generate request arrival traces based on a Poisson distribution with a mean rate of 1 request per second.

4.2 Context Switching Trace Simulation

As there are no publicly available context switching traces for Large Language Model as a Service (LLMaaS) workloads, we simulate two patterns of context switching to assess system behavior under different conditions. These simulations include:

Random: In this pattern, context switching occurs unpredictably, with priority changes happening arbitrarily. There is no temporal correlation, simulating a dynamic and uncontrolled environment where the workload characteristics are highly unpredictable.

Markov: This pattern introduces temporal locality into the context switching process. Requests that have been frequently or recently served are given higher priority, reflecting a more structured scenario where the workload exhibits some continuity or locality in its usage patterns.

In both cases, the priorities are determined offline, meaning they are precomputed based on the simulation patterns rather than dynamically adjusted during runtime. However, our design, FastSwitch, does not rely on these offline priority predictions. It remains flexible and adaptive, allowing for dynamic scheduling based on the actual runtime demands of the specified use case. This design choice ensures that FastSwitch can efficiently handle a wide range of context switching behaviors, from highly unpredictable scenarios to those that follow more predictable patterns.

4.3 Baselines and Metrics

We compare FastSwitch with vLLM. The key metrics used for evaluation include the P95, P99, and P99.9 TTFT, which measure the latency experienced by the 95th, 99th, and 99.9th percentiles of requests before the first token of each turn is generated. In addition, we evaluate the P99.9 TBT, which captures the latency between consecutive tokens in the generated responses. Furthermore, we compare the end-to-end throughput of both systems. These metrics offer a comprehensive view of the system's performance, especially under different load conditions and priority schemes.

4.4 Implementation

FastSwitch is built on vLLM with over 5,000 lines of Python and 1,000 lines of CUDA/C++ code. Utilizing "prefill with prefix" triton kernel from lightllm(ModelTC, 2024), FastSwitch supports multi-turn conversations. We developed a priority-based scheduler that enhances existing scheduling policies by dynamically adjusting priorities and managing requests queues in real-time. Based on the priority-update frequency, at the iteration where a priority change occurs, the scheduler reorders requests across waiting, running, and swapped queues to meet the updated priority requirements. During other iterations, it adheres to the most recently updated prioritiy to handle scheduling and service execution.

5 EVALUATION

5.1 End-to-End Performance

We evaluate the end-to-end performance of FastSwitch by comparing it with the baseline under appropriate priority-update frequency. For Qwen-32B, we set the priority-update frequency to 0.02 (once every 50 iterations), following the study in Andes(Liu et al., 2024a) to maximize the quality of experience (QoE) for the round-robin pattern. On the other hand, for a smaller model LLaMA-8B, we double the priority-update frequency to 0.04 (once every 25 iterations) to better highlight the optimizations in context switching achieved by our design, especially under more frequent context switching.

5.1.1 Latency Metrics

We begin by analyzing latency metrics, including P95, P99, P99.9 TTFT, and P99.9 TBT, for both the LLaMA-8B and Qwen-32B models under Markov and Random context switching patterns using FastSwitch.

As illustrated in Figure 8 (a)–(b), for the LLaMA-8B model, FastSwitch demonstrates substantial improvements in latency across both context switching patterns. Under the Markov trace, FastSwitch achieves speedups of 5.8×, 4.1×, 3.7×, and 2.0× for P95 TTFT, P99 TTFT, P99.9 TTFT, and P99.9 TBT, respectively. When evaluated with the Random trace, FastSwitch attains speedups of 4.3×, 3.7×, 2.5×, and 2.7× for the same metrics. These results indicate that FastSwitch effectively manages temporal locality in the Markov trace while maintaining robust performance under unpredictable priority changes in the Random trace.

As illustrated in Figure 8 (c)–(d), for the Qwen-32B model, FastSwitch exhibits considerable performance enhancements as well. In the Markov scenario, FastSwitch achieves speedups of 1.7×, 1.6×, 1.4×, and 11.2× for P95 TTFT, P99 TTFT, P99.9 TTFT, and P99.9 TBT, respectively. Under the Random trace, FastSwitch exhibits speedups of 1.4×, 1.5×, 1.4×, and 3.6× for the same metrics.

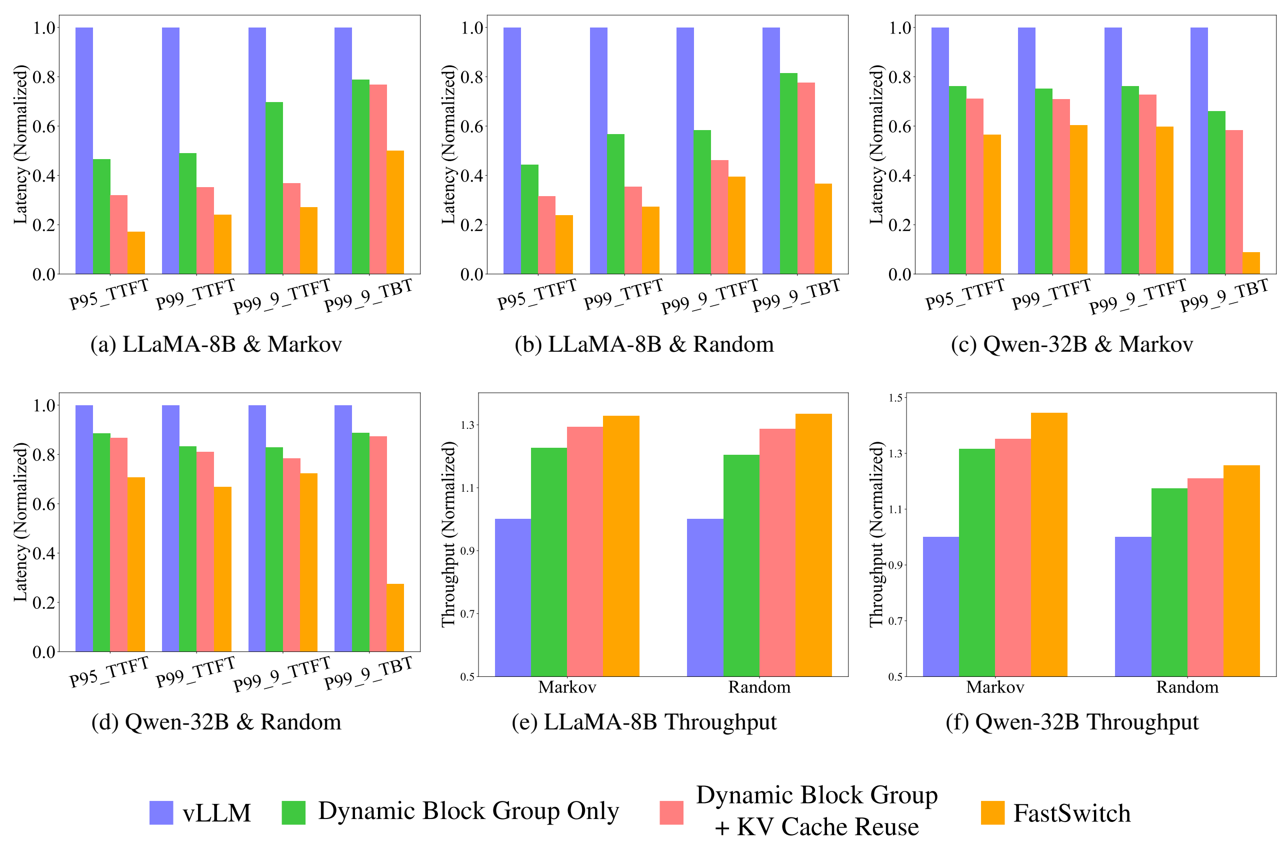

Figure 8. Comparison of TTFT, TBT, and throughput between FastSwitch and baseline under different models and traces.

5.1.2 *End-to-End Throughput Improvement*

In addition to latency metrics, we evaluate the end-to-end throughput of FastSwitch under varying priority-update frequencies for both models. As illustrated in Figure 8 (e)–(f), FastSwitch consistently enhances throughput across different priority-update frequencies.

For the LLaMA-8B model, FastSwitch achieves up to a 1.334× increase in throughput under high priority-update frequency across both patterns, maintaining efficient token generation without significant delays. Similarly, the Qwen-32B model experiences up to a 1.444× improvement in throughput. The larger throughput gains observed for Qwen-32B are attributed to its higher swapping latency compared to its inference time, which FastSwitch effectively mitigates through the three optimizations.

5.1.3 *Summary*

The contributions of each sub-design in FastSwitch, are critical in reducing latency and improving throughput. FastSwitch consistently outperforms other configurations across different models and context switching patterns, demonstrating its scalability and effectiveness in managing complex workloads.

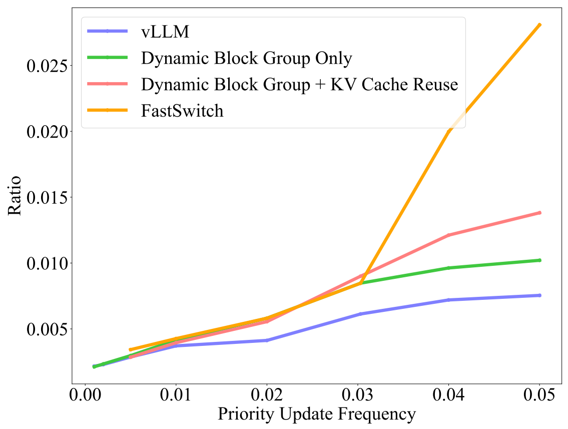

Figure 9. Show the call stack overhead after applying each optimization.

5.2 Call Stack Overhead Analysis

Figure 9 illustrates the call stack overhead as priority-update frequency increases. Each part of FastSwitch introduces optimizations that slightly raise overhead but improve performance. Despite the gradual increase, the overhead contributes to no more than a 1 % rise in end-to-end time. This indicates a minimal impact on overall system efficiency. As the frequency increases, the call stack overhead also increases. This is due to the need to resolve more KV cache conflicts and synchronize more ongoing swap-in requests dispatched in pass iterations by the end of the schedule. This results in some context switching overhead being added to the call stack overhead. However, pure call stack overhead remains within 1 %.

5.3 Breakdown and Sensitivity Analysis

To gain deeper insights into the effectiveness of our proposed system and its sensitivity to various parameters, we perform a series of breakdown analysis experiments using the LLaMA-8B model on the ShareGPT dataset, running on a single A10 GPU. The average request rate is set to 1.0 req/s.

5.3.1 Dynamic Block Group Manager

Effectiveness of the Dynamic Block Group Manager. The Dynamic Block Group Manager employs a coarse-grained KV cache allocation approach, resulting in simplified swapping operations and reduced context switching overhead, as demonstrated by the ratio of context switching overhead to end-to-end latency shown in Figure 10. The coarse-grained approach shows up to 3.11× context switching speedup compared to the vLLM baseline across various frequencies.

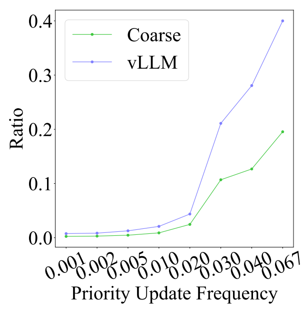

Figure 10. Context switching overhead across priority-update frequencies.

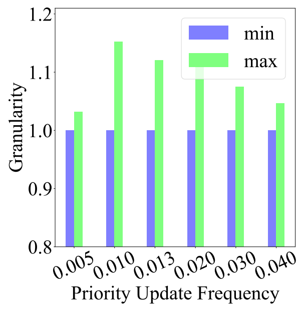

Figure 11. Sensitivity study.

Initial Dynamic Block Group Size Sensitivity. We set the initial block group size to 1,000 tokens, or about 70 vLLM blocks. A sensitivity analysis, shown in Figure 11, examines the average swap in and swap out granularity across initial block group sizes (64 to 3,000 tokens) and varying priority update frequencies. The values are normalized, with the minimum set to 1. The results show that for a fixed priority-update frequency, changing the initial block group size causes no more than a 15.13% difference in granularity. This indicates that the system is robust to variations in block group size. Granularity is mainly influenced by the GPU memory allocated to the KV cache for each task, making GPU memory a key factor in swapping efficiency.

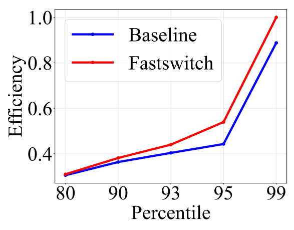

Figure 12. Efficiency comparison.

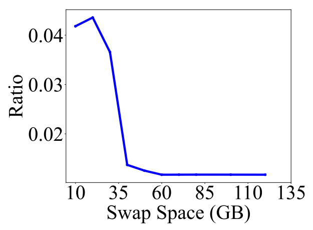

Figure 13. Sensitivity study.

5.3.2 Multithreading Swap Manager

Token Generation Efficiency. The asynchronous swapping facilitated by the Multithreading Swap Manager improves token generation efficiency. In Figure 12, we compare the token generation efficiency between the baseline and FastSwitch. After introducing the Multithreading Swap Manager, the system performs more iterations with reduced latency. However, the increased number of iterations complicates direct comparison with the baseline. To address this, we divided the inference process into fixed-iteration intervals, with an interval size of 5 iterations, calculating the number of new tokens generated and the time taken within each interval. This allows us to determine the token generation efficiency per unit of time for each interval. We then compared the token generation efficiency across different quantiles for both the baseline and the Multithreading Swap Manager. The results show that FastSwitch consistently achieves higher token generation efficiency across almost all quantiles, with particularly significant improvements at higher percentiles - showing a 21.8% increase at P99 and a 12.6% increase at P99.9 compared to the baseline. These results demonstrate the effectiveness of the Multithreading Swap Manager in minimizing the impact of KV cache transfers on overall throughput.

Table 1. Microbenchmark comparison.

Metric	Traditional Swap Out	Optimized Swap Out with KV Cache Reuse
Num blocks	122030	58187
Num operations	13076	10713
Granularity	9.3	5.43
Latency	15.5s	6.7s

5.3.3 KV Cache Reuse Mechanism

Reduction in swap Out Size. The KV Cache Reuse Mechanism significantly reduces the total number of swap out blocks by 53%, as shown in Table 1, directly correlating with lower preemption latency and stalled time.

Sensitivity of CPU Memory Size on Context Switching Overhead. We evaluated how varying CPU memory sizes for KV cache copies impact the KV Cache Reuse Mechanism by measuring context switching overhead. As shown in Figure 13, increasing memory allows less KV cache copies to be contaminated, reducing context switching overhead by enabling greater cache reuse and minimizing redundant KV cache swapping

Our findings show that larger memory allocation reduce overhead by supporting more cache reuse across conversation turns. However, beyond 60 GB, further increases yield diminishing returns, suggesting 60 GB as an optimal allocation for this setup.

6 Related Work

6.1 Scheduling, SLOs, and Fairness in LLM Serving

Maintaining fairness and meeting SLOs in LLM serving systems is crucial for performance. Sheng et al. (Sheng et al., 2024) proposed VTC, a fair scheduler for LLM serving that handles unpredictable request lengths and dynamic batching, ensuring fairness at the token level. Liu et al. (Liu et al., 2024a) proposed Andes, which balances latency and quality in LLM-based text streaming services through optimized scheduling. Wu et al. (Wu et al., 2023) introduced a Fast Distributed Inference Serving system that improves scheduling and resource management in distributed environments.

6.2 KV Cache Management

KV cache (Liu et al., 2024c; Qin et al., 2024; Ge et al., 2023) stores precomputed key-value projections from previous tokens during Transformer model inference. Efficient KV cache management accelerates LLM inference, particularly in multi-call and batched executions. The cache hold intermediate key-value pairs for self-attention (Shaw et al., 2018), enabling reuse of computations during token generation. vLLM (Kwon et al., 2023) employs paging for memory-efficient KV cache management, enabling large batch processing through dynamic GPU memory paging. SGLang (Zheng et al., 2023) introduces RadixAttention, organizing KV cache in a radix tree for systematic cache reuse across shared-prefix requests, with LRU-based eviction. Unlike vLLM's batch-level focus, RadixAttention handles both intra-program parallelism and multi-call workflows.

Other systems like TensorRT-LLM (NVIDIA, 2024) and Hugging Face Accelerate (Hugging Face, 2024) optimize throughput via dynamic batch sizing, but lack fine-grained prefix reuse and intra-program parallelism capabilities found in vLLM and SGLang.

7 Conclusion

We present FastSwitch, a serving system that optimizes scheduling in LLM inference through advanced KV cache management and preemption design. FastSwitch enhances preemptive context switching in LLM serving systems, minimizing performance penalties in terms of TTFT, TBT, and throughput.

Compared to vLLM, evaluations show FastSwitch delivers various percentiles of TTFT speedups of 1.4-5.8×, P99.9 TBT speedup of up to 11.2×, and the throughput increase of up to 1.44×. These gains highlight FastSwitch's ability to ensure fairness and enhance user experience in dynamic, large-scale LLM deployments.

In conclusion, FastSwitch provides an efficient, solution for preemptive scheduling, reducing latency and boosting throughput while maintaining fairness.

References

Agrawal, A., Panwar, A., Mohan, J., Kwatra, N., Gulavani, B. S., and Ramjee, R. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369*, 2023.

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. *arXiv preprint arXiv:2403.02310*, 2024.

Aminabadi, R. Y., Rajbhandari, S., Zhang, M., Awan, A. A., Li, C., Li, D., Zheng, E., Rasley, J., Smith, S., Ruwase, O., and He, Y. Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale, 2022. URL https://arxiv.org/abs/2207.00032.

Bisk, Y., Zellers, R., Gao, J., Choi, Y., et al. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pp. 7432–7439, 2020.

Brown, T. B. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

Gao, B., He, Z., Sharma, P., Kang, Q., Jevdjic, D., Deng, J., Yang, X., Yu, Z., and Zuo, P. Attentionstore: Cost-effective attention reuse across multi-turn conversations in large language model serving. *arXiv preprint arXiv:2403.19708*, 2024.

Ge, S., Zhang, Y., Liu, L., Zhang, M., Han, J., and Gao, J. Model tells you what to discard: Adaptive kv cache compression for llms. *arXiv preprint arXiv:2310.01801*, 2023.

Gong, C., He, D., Tan, X., Qin, T., Wang, L., and Liu, T.-Y. Frage: Frequency-agnostic word representation. *Advances in neural information processing systems*, 31, 2018.

Hendrycks, D., Burns, C., Basart, S., Zou, A., Mazeika, M., Song, D., and Steinhardt, J. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*, 2020.

Hugging Face. Hugging face large language models (llms). https://huggingface.co/, 2024. Accessed: 2024-10-28.

Koh, J. Y., Fried, D., and Salakhutdinov, R. R. Generating images with multimodal language models. *Advances in Neural Information Processing Systems*, 36, 2024.

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*, pp. 611–626, 2023.

Larimi, S. S. N., Salami, B., Unsal, O. S., Kestelman, A. C., Sarbazi-Azad, H., and Mutlu, O. Understanding power consumption and reliability of high-bandwidth memory with voltage underscaling, 2020. URL https://arxiv.org/abs/2101.00969.

Liu, J., Wu, Z., Chung, J.-W., Lai, F., Lee, M., and Chowdhury, M. Andes: Defining and enhancing quality-of-experience in llm-based text streaming services, 2024a. URL https://arxiv.org/abs/2404.16283.

Liu, N., Chen, L., Tian, X., Zou, W., Chen, K., and Cui, M. From llm to conversational agent: A memory enhanced architecture with fine-tuning of large language models, 2024b. URL https://arxiv.org/abs/2401.02777.

Liu, Y., Li, H., Cheng, Y., Ray, S., Huang, Y., Zhang, Q., Du, K., Yao, J., Lu, S., Ananthanarayanan, G., et al. Cachegen: Kv cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pp. 38–56, 2024c.

Minaee, S., Mikolov, T., Nikzad, N., Chenaghlu, M., Socher, R., Amatriain, X., and Gao, J. Large language models: A survey. *arXiv preprint arXiv:2402.06196*, 2024.

ModelTC. Lightllm: A lightweight framework for large language model inference. https://github.com/ModelTC/lightllm, 2024. A Python-based LLM inference and serving framework with lightweight design, easy scalability, and high-speed performance.

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., and Xiong, C. Codegen: An open large language model for code with multi-turn program synthesis, 2023. URL https://arxiv.org/abs/2203.13474.

NVIDIA. Nvidia tensorrt-llm. https://docs.nvidia.com/tensorrt-llm/index.html, 2024. Accessed: 2024-10-28.

OpenAI. Chatgpt. https://openai.com/chatgpt, 2024. Accessed: 2024-10-28.

Patke, A., Reddy, D., Jha, S., Qiu, H., Pinto, C., Cui, S., Narayanaswami, C., Kalbarczyk, Z., and Iyer, R. One queue is all you need: Resolving head-of-line blocking in large language model serving. *arXiv preprint arXiv:2407.00047*, 2024.

Qin, R., Li, Z., He, W., Zhang, M., Wu, Y., Zheng, W., and Xu, X. Mooncake: Kimi's kvcache-centric architecture for llm serving. *arXiv preprint arXiv:2407.00079*, 2024.

ShareGPT. Sharegpt: Share your wildest chatgpt conversations with one click. https://sharegpt.com/, 2024.

Shaw, P., Uszkoreit, J., and Vaswani, A. Self-attention with relative position representations. *arXiv preprint arXiv:1803.02155*, 2018.

Sheng, Y., Zheng, L., Yuan, B., Li, Z., Ryabinin, M., Chen, B., Liang, P., Ré, C., Stoica, I., and Zhang, C. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pp. 31094–31116. PMLR, 2023.

Sheng, Y., Cao, S., Li, D., Zhu, B., Li, Z., Zhuo, D., Gonzalez, J. E., and Stoica, I. Fairness in serving large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 965–988, 2024.

Sun, B., Huang, Z., Zhao, H., Xiao, W., Zhang, X., Li, Y., and Lin, W. Llumnix: Dynamic scheduling for large language model serving, 2024. URL https://arxiv.org/abs/2406.03243.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Von Puttkamer, E. A simple hardware buddy system memory allocator. *IEEE Transactions on Computers*, C-24 (10):953–957, 1975. doi: 10.1109/T-C.1975.224100.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Wu, B., Zhong, Y., Zhang, Z., Huang, G., Liu, X., and Jin, X. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920*, 2023.

Yang, A., Yang, B., Hui, B., Zheng, B., Yu, B., Zhou, C., Li, C., Li, C., Liu, D., Huang, F., et al. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*, 2024.

Yin, W., Xu, M., Li, Y., and Liu, X. Llm as a system service on mobile devices. *arXiv preprint arXiv:2403.11805*, 2024.

Zheng, L., Yin, L., Xie, Z., Huang, J., Sun, C., Hao Yu, C., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., et al. Efficiently programming large language models using sglang. *arXiv e-prints*, pp. arXiv–2312, 2023.

Zhu, W., Liu, H., Dong, Q., Xu, J., Huang, S., Kong, L., Chen, J., and Li, L. Multilingual machine translation with large language models: Empirical results and analysis, 2024. URL `https://arxiv.org/abs/2304.04675`.