FastSwitch: Optimizing Context Switch Efficiency in Resource Intensive Large Language Model Scheduling

Anonymous Authors[1]

Abstract

Recent advancements in Large Language Models (LLMs) have transformed various applications, necessitating efficient inference systems to handle high-demand scenarios with varying priorities. Preemption-based scheduling ensures fairness by dynamically adjusting task priorities, but frequent context switching introduces overhead, degrading Time-to-First-Token (TTFT), Time-Between-Tokens (TBT), and system throughput. We introduce FastSwitch, a novel system that mitigates the overhead of frequent context switching in LLM inference while preserving fairness, through innovations like a Dynamic Block Group Manager for efficient GPU memory management with larger granularity, a Multi-threaded Swap Manager for asynchronous swap handling, and a KV cache reuse mechanism for optimizing preemption in multi-turn conversations. Our evaluation with LLaMA 8B and Qwen 32B on NVIDIA A10 and A100 GPUs shows that FastSwitch achieves a speedup of 1.4×–5.8× in TTFT across percentiles, with up to 5.8×, 4.3×, and 3.7× improvements at p95, p99, and p99.9, respectively, alongside a p99.9 TBT improvement of up to 11.2× and a throughput increase of up to 1.44×. These results demonstrate FastSwitch's effectiveness in maintaining fairness, performance, and throughput, 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, deploying LLMs poses significant challenges due to their high computational and memory requirements. Efficient execution typically relies on high-end GPUs with substantial high-bandwidth memory (HBM) (Larimi et al., 2020), but the associated costs and limited availability hinder scalability, especially in resource-constrained settings. Additionally, 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 escalating resource demands.

To address these challenges, many LLM serving systems prioritize fairness by dynamically adjusting request priorities during runtime based on Service Level Objective (SLO) metrics. Given that GPU memory resources are always constrained, preemption mechanisms become essential to efficiently manage resources and maintain 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 preemption scheduling to ensure fairness and address 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. Analysis of ShareGPT (ShareGPT, 2024) in Figure 4 reveals that 78% of conversations are multi-turn, and 72% of conversations exceed 4K tokens, necessitating frequent context switching.

However, frequent priority changes can lead to numerous context switching, requiring the transfer of large Key-Value (KV) cache between GPU and CPU memory as most sys-

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the Machine Learning and Systems (MLSys) Conference. Do not distribute.

tems (Kwon et al., 2023; Zheng et al., 2023) use swapping as the default preemption approach. This transfer incurs significant overhead, which degrades key performance metrics such as Time-to-First-Token (TTFT) and Time-Between-Tokens (TBT), negatively impacting the overall Quality of Experience (QoE) (Liu et al., 2024a). As shown in Figure 1, the stall time caused by preemption can take several times longer than the execution time in a single iteration.

Addressing the overhead from frequent context switching is crucial for enhancing the performance and scalability of LLM serving systems. Although previous works have proposed strategies to mitigate this overhead (Sun et al., 2024; Gao et al., 2024; Wu et al., 2023), challenges remain in balancing fairness, throughput, and performance penalties.

Existing approaches, such as Llumnix (Sun et al., 2024), allocate buffers to merge KV cache before transmission and perform a secondary transfer to achieve larger transmission granularity. However, this requires additional memory buffers for support, which may violate vLLM's memory management principles and introduce overhead from a second transmission. AttentionStore (Gao et al., 2024) reuses the historical KV cache for pre-filling and supports layer-wise asynchronous processing. Similarly, FastServe (Wu et al., 2023) employs an interation-wise asynchronous transmission and prefetching mechanism to predict KV cache usage and pre-load it. However, the layer-wise asynchronous method can interfere with existing graph execution strategies and introduce additional decoding latency. We will compare these works with ours in Section 2.3.

In response, we introduce FastSwitch, a novel system that optimizes preemptive context switching in LLM inference with minimal additional cost. FastSwitch directly 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:

- Dynamic Block Group Manager: Inspired by buddy allocation, this component allocates memory at a coarser granularity. It reduces call stack overhead from memory copy kernel dispatch and optimizes bandwidth during KV cache transfers, while the Multi-threaded Swap Manager asynchronously manages KV cache transfers, minimizing delays from cache dependencies during context switching and improving token generation efficiency. Additionally, the KV Cache Reuse Mechanism reuses KV cache across conversation turns in multi-turn dialogues.
- In-depth Evaluation: Demonstrate substantial improvements in TTFT (with a speedup of up to 1.4×–5.8×), TBT (with a speedup of up to 11.2×), and Throughput (with an increase of up to 1.44×), on NVIDIA GPUs compared to vLLM, particularly under high priority-update frequency.
- 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 reallocates resources dynamically. There are two main preemption methods: **recomputation-based preemption**, which halts a task and recomputes its KV cache upon resumption, increasing latency and resource use, especially for long contexts; and **swap-based preemption**, which transfers the KV cache between GPU and CPU memory, avoiding recomputation. Systems like vLLM effectively use swap-based preemption to manage memory. 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 approach builds on vLLM by optimizing its swap-based preemption.

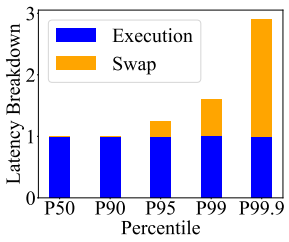

Figure 1. The execution time and additional stall time due to swap operations across percentiles.

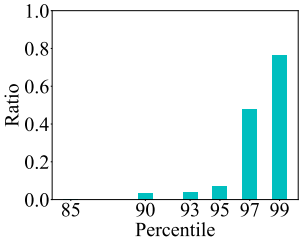

Figure 2. In each iteration, only a small ratio of requests to be executed need to wait for the KV cache.

2.2 Motivation

2.2.1 *Coarse-Grained Memory Management*

Observations 1: Context Switching Overheads Preemptive context switching, necessary for managing dynamic workloads and prioritizing tasks, introduces significant over-

heads, 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 = 1 req/s and priority updates every 100 iterations. We normalized the latency by setting the execution time of decoding to 1. The latency penalty from preemption was measured as KV cache swap time.

Figure 1 shows that p99 latency is approximately 1.6 times higher than the p50, with swap-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 decoding time.

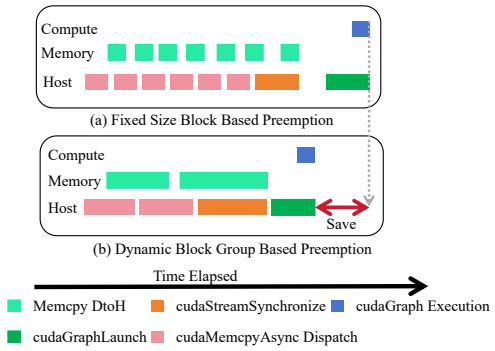

Figure 3. Timeline comparison of fixed-size block based preemption and dynamic block group based preemption.

Observations 2: Necessity of Granularity Improvements While vLLM's PagedAttention reduces internal fragmentation by using non-contiguous virtual memory for KV cache, optimizing memory transfer patterns remains a challenge. As shown in Figure 3(a), performance bottlenecks arise from suboptimal memory transfer granularity like small (128 KB) KV cache transfers in LLaMA-8B. The dispatch overhead for each `cudaMemcpyAsync()` exceeds its 10 µs runtime, causing GPU idle time. This inefficiency is worsened by the transfer size being below PCIe 4.0's optimal 320 KB.

Solution 1: Instead of handling individual blocks, managing memory in larger granularities helps maintain memory continuity and reduce context switching overhead. This approach enables better utilization of PCIe bandwidth during KV cache transfers and minimizes call stack overhead.

2.2.2 *Multi-threaded Swap Management*

Observation 3: Limited Impact of KV Cache Transfers on Majority of Requests 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 Figure 2, the impact of global priority updates across requests is most pronounced in tail cases where a high proportion of requests experience delays. However, in most scenarios, KV cache transfers between CPU and GPU memory affect only a subset of requests. Though most requests proceed uninterrupted, preemption overhead can exceed a single execution step, potentially degrading performance.

Solution 2: In each iteration, only a subset of requests need KV cache transfers, while most proceed without them. Preemption costs, often exceeding the time of a single execution step, are primarily due to KV cache swap latency. By handling transfers asynchronously and using multi-threaded swap management, the system overlaps swap and compute, improving token generation efficiency. This approach reduces context switching latency, prioritizes high-importance tasks, and maintains high throughput by selectively managing swaps based on real-time workload, minimizing the performance impact of swap-related delays.

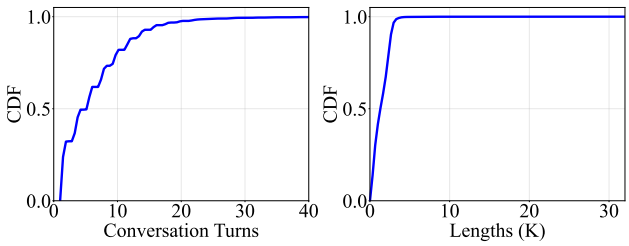

Figure 4. ShareGPT Conversation Turns & Lengths Distribution.

2.2.3 *KV Cache Reuse in Multi-turn Conversations*

Observation 4: Prevalence of Multi-turn Conversations Analysis of the ShareGPT dataset (ShareGPT, 2024) (100K conversations) shows 78% of interactions are multi-turn, averaging 5.5 turns per conversation (Figure 4). These multi-turn conversations often involve lengthy contexts, leading to large Key-Value (KV) cache that is maintained to ensure coherent context across turns. Given the number of turns and the extended context, there is a high percentage of repeated KV cache computations. This presents opportunities for KV cache reuse, which can significantly improve efficiency by reducing redundant computations.

Solution 3: Every time preemption occurs, having the historical KV cache stored in CPU memory reduces the amount of data that needs to be swapped out to free up GPU memory. This reuse minimizes redundant KV transfers, saving time

and improving efficiency. By retaining relevant KV cache in CPU, the system can significantly reduce preemption overhead and improve token generation speed in multi-turn conversations.

Table 1. Features Comparison Across Different Methods

Features	Llumnix	Attentionstore	Fastserve	FastSwitch
I/O Aware	✓	✗	✗	✓
KV Cache with Zero Memory Waste	✗	✓	✓	✓
Async Computation & Swapping	✓	✓	✓	✓
KV Cache Reuse	✗	✓	✗	✓
cudaGraph Execution	✓	✗	✓	✓

2.3 Comparison to Prior Works

Llumnix merges KV cache before transmission to increase granularity, but this requires extra buffers, potentially violating vLLM's memory management and adding overhead. AttentionStore reuses historical KV cache for pre-filling and supports layer-wise asynchronous processing, but this can interfere with graph execution and add decoding latency given that preemption latency might exceed decoding latency. Also, it has explored KV cache reuse by leveraging prefix reuse in multi-turn conversations. However, it considers storage tiers beyond main memory, such as SSDs. In contrast, our method focuses exclusively on GPU and CPU memory, optimizing within this more constrained environment. Additionally, unlike prior works that primarily target swap-in processes for prefill, our approach is designed specifically for preemption and swap-out operations. We introduce a mechanism to handle partial cache contamination where cache in CPU is used by higher priority requests, enabling the reuse of partially contaminated KV cache while minimizing preemption overhead. FastServe uses iteration-wise asynchronous transmission to predict and pre-load KV cache while supporting CUDA Graph execution for GPU efficiency. However, predicting appropriate requests used for preemptive swap-out can be challenging. Last but not least, none of the previous works have addressed the issue of dispatch overhead during context switching. In summary, FastSwitch optimizes I/O utilization with no memory waste in KV cache, supports asynchronous computation, reuses KV cache, and minimizes context switching overhead, improving fairness, throughput, and scalability without extra memory or latency.

3 Design of FastSwitch

As shown in Figure 5, FastSwitch enhances preemptive context switching efficiency in LLM serving. To address the challenges outlined in Section 2.2, FastSwitch comprises three key components that work in concert to optimize performance and resource utilization. The **Dynamic Block Group Manager** maximizes PCIe bandwidth utilization through larger-granularity memory units, effectively addressing the challenges of dynamic KV cache allocation and call stack overhead. This component also handles memory conflict resolution and manages KV cache tracking and reuse. Building on this foundation, the **Multi-Threaded Swap Manager** leverages async swap operations to improve token generation throughput. Finally, the **Priority Scheduler** schedules high-priority requests into the running batch based on the latest priorities.

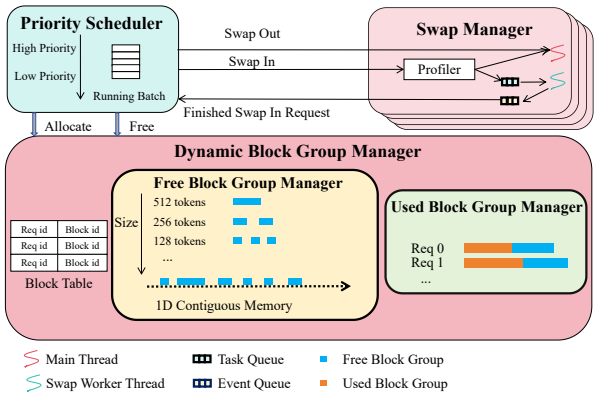

Figure 5. FastSwitch system overview.

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

Managing GPU memory effectively is critical for high-throughput inference. Simply increasing the block size can lead to internal fragmentation, resulting in inefficient memory usage and undermining the memory utilization efficiency that vLLM aims to achieve. This is why vLLM sets the default block size to 16 tokens, balancing memory efficiency with performance. Traditional approaches that reserve KV cache memory ahead-of-time exacerbate fragmentation, while dynamic allocation introduces complexities and overheads in context switching. Developing lightweight memory management strategies that integrate seamlessly with vLLM's existing policies is essential to balance efficient memory utilization with the need for dynamic and flexible allocation.

Efficient GPU memory management is crucial for high-throughput inference in Large Language Models (LLMs). LLM serving systems, such as vLLM, use fixed KV cache allocations to reduce memory waste. However, preemption through swapping is limited by block granularity. As mentioned in Section 2.2.1, dispatch overhead becomes a bottleneck, especially as context lengths and switching frequencies increase, leading to a substantial impact on I/O utilization.

To address these challenges, we propose the Dynamic Block Group Manager, a novel memory management component inspired by the buddy allocation system. This manager is designed to increase the granularity and size of the KV cache allocation, thereby reducing dispatch time and enhancing I/O bandwidth utilization. By leveraging the principles of the buddy allocation system, the Dynamic Block Group Manager ensures that each request is allocated memory blocks of the size it requires, optimizing transfer efficiency.

1. **Dynamic Block Group Allocation and Management**: Inspired by the buddy allocation system, the Dynamic Block Group Manager organizes memory through two primary subcomponents: the free block manager and the used block manager. Memory is allocated in larger chunks known as block groups, each comprising multiple contiguous blocks. When no free block group is available, the manager dynamically splits a used block group that is not fully allocated by other requests, ensuring efficient memory utilization without waste.

   To maintain optimal memory usage, the manager supports dynamic splitting and merging of block groups. When a request's memory requirement does not fully utilize a block group, the manager can split the block group into smaller units, reallocating the unused portions 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.

2. **Optimized Transfer Granularity and Enhanced Bandwidth Utilization**: 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. Larger memory chunks in single operations maximize PCIe bandwidth use, improving data throughput and minimizing idle resources. As shown in Figure 3(b), compared to fixed size blocks, our design consolidates small memory operations into fewer, larger transfers, reducing dispatch overhead and improving PCIe utilization.

3. **Integration with Existing Memory Management**: The Dynamic Block Group Manager integrates with vLLM's memory management, complementing PagedAttention's dynamic allocation. It maximizes KV cache continuity without waste, maintaining compatibility with existing LLM frameworks.

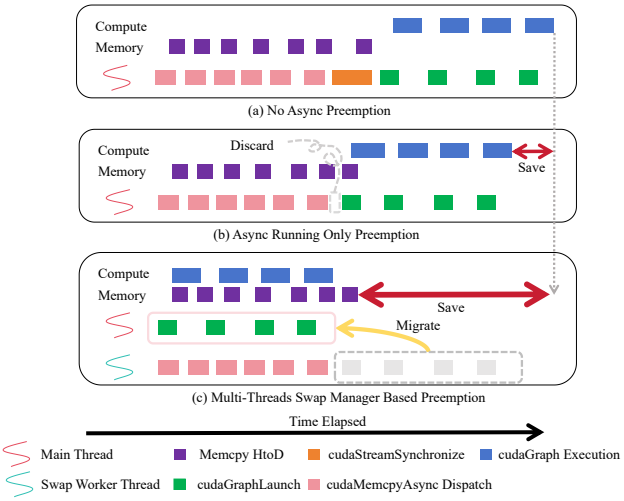

Figure 6. Comparison of varying degrees of asynchronous preemption.

3.2 Multi-threaded Swap Manager for Optimizing Token Generation Efficiency

As discussed in Section 2, since only a small fraction of requests require KV cache transfers, most active requests proceed without delays. This allows overlapping swap operations with other active requests through asynchronous handling. By implementing a fully asynchronous, multi-threaded swap management system, FastSwitch enhances token generation efficiency, minimizes stalls, and ensures high throughput and responsiveness under dynamic, high-demand workloads.

Managing Memory Conflicts in Asynchronous Execution: Asynchronous KV cache transfers allow overlapping swap and compute operations, enhancing token generation efficiency. However, this introduces potential memory allocation conflicts between concurrent KV transfer tasks and active inference operations. Effective synchronization mechanisms are essential to prevent resource contention and maintain performance.

Overcoming Python GIL Limitations: Python-based call stacks in many inference frameworks introduce the Global Interpreter Lock (GIL), which bottlenecks parallel execution of asynchronous tasks by limiting CPU-side kernel dispatch. While GPU kernel execution remains unaffected, the restricted dispatch diminishes the benefits of asynchronous optimizations and constrains overall system throughput and scalability.

To address these challenges, the Multi-threaded Swap Manager employs several strategies to optimize token generation efficiency:

1. **Adaptive Swap Strategy**: The Swap Manager em-

ploys dynamic waiting based on the system's current state. It continuously monitors key metrics such as the number and size of ongoing swap operations over a recent time window. A profiler records this information to enable informed decision-making. While asynchronous swap-ins can typically reduce idle time and improve efficiency, they are not always the optimal approach. Specifically, when the total number of requests is high but each request is relatively short, asynchronous swap-ins may degrade token generation efficiency. In such cases, it is more beneficial to perform synchronous swap-ins, as the overhead of the swap is minimal compared to the potential gain in processing a larger number of tokens. This adaptive strategy carefully balances swap overhead with token generation efficiency, dynamically switching between asynchronous and synchronous swap-ins based on real-time workload characteristics.

2. **Resource Tracking and Conflict Resolution**: The Dynamic Block Group Manager monitors memory resource allocation and usage, enabling the Swap Manager to synchronize KV cache transfers effectively. This minimizes resource contention and resolves memory conflicts, ensuring efficient operation during overlapping swap and inference tasks.

3. **Asynchronous Execution Handling**: By allowing certain swap operations to occur in parallel with active inference processes, the Swap Manager reduces idle times and enhances overall throughput. This asynchronous handling ensures that the majority of requests can proceed without being delayed by swap 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 execution times.

   In (b), only the execution 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 of `cudaMemcpyAsync` are asynchronous. 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 Multi-threaded Swap Manager within FastSwitch. The algorithm efficiently manages swap operations while addressing potential memory conflicts and dynamically adjusting its execution strategy. The process starts by monitoring ongoing swap-in operations. Once a swap-in is completed, the corresponding request is moved from the `swapped` queue to the `running` queue, and the `recent_swap_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 swapped-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 swap metrics from the `recent_swap_info` queue to determine whether to proceed with the current iteration or to initiate a synchronization of the swap-in stream.

Algorithm 1 Multi-threaded Swap Manager Algorithm

```
Initialization:
  running ← InitializeQueue()
  swapped ← InitializeQueue()
  ongoing_swapped_in ← InitializeQueue()
  recent_swap_info ← InitializeQueue()
for each iteration do
  Step 1: Verify Swap-In Completion
  for each req in ongoing swapped in do
    if IsSwapInCompleted(req) then
      Move(req, ongoing_swapped_in, running)
    end if
  end for
  Step 2: Handle Swap-In Requests
  if HasSwapInRequests(swapped) then
    ExecuteSwapInOperations(swapped)
    UpdateQueue(recent_swap_info, swapped, "SwapIn")
  end if
  Step 3: Handle Swap-Out Requests
  if HasSwapOutRequests(running) then
    ExecuteSwapOutOperations(running)
    UpdateQueue(recent_swap_info, running, "SwapOut")
    Step 3a: Conflict Detection
    if DetectConflict(running, swapped) then
      Synchronize(running, swapped)
    end if
  end if
  Step 4: Dynamic Execution Strategy
  strategy ← DetermineExecutionStrategy(running, swapped,
  recent_swap_info)
  if strategy = "yes" then
    SwapInStreamSynchronize()
  else
    MovePendingTasks(running, ongoing_swapped_in)
  end if
end for
```

3.3 KV Cache Reuse Mechanism for Efficient Multi-turn Conversations

Building upon the observations outlined in Section 2.2.3, particularly the inefficiencies arising from repetitive Key-Value (KV) cache computations during multi-turn conversations, we introduce the KV Cache Reuse Mechanism. This mechanism optimizes KV cache management, particularly in scenarios involving swap-outs during preemption, by leveraging KV cache copies stored in CPU memory to minimize the amount of transferred data.

However, given that our experimental setup and assumptions involve only CPU and GPU resources, and CPU memory is not unlimited, a cached copy of a request in CPU memory may be overwritten by higher-priority tasks. This presents a challenge where the cached copy could become unusable. To address this, we designed an algorithm to effectively manage this issue.

Additionally, the mechanism preallocates space for the swap-out increment at the conclusion of the next conversation, ensuring better memory continuity and reducing fragmentation. This design integrates seamlessly with the Dynamic Block Group Manager to facilitate efficient memory allocation and management.

1. **KV Cache Reuse Strategy**: Our strategy focuses on retaining and efficiently managing KV cache by keeping a copy of the KV cache in CPU memory. This approach reduces the amount of data that needs to be swapped out during preemption, 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.

2. **Selective KV Cache Swapping and Preallocation**: 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 data transferred between CPU and GPU memory and reducing preemption latency. Furthermore, the system preallocates additional memory space for the next iteration's swap-out increment, based on the KV cache copy already stored in CPU memory. This proactive allocation improves memory continuity, prevents fragmentation, and ensures smoother transitions between iterations.

3. **Contamination Tracking and Handling**: To ensure the integrity 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. Only uncontaminated block groups are eligible for reuse, preventing erroneous data access and maintaining cache integrity.

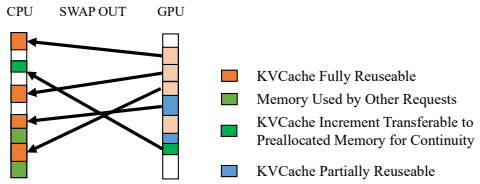

Figure 7. Workflow of the KV Cache Reuse Mechanism.

4 METHODOLOGY

We evaluate FastSwitch using LLaMA 8B and Qwen 32B models on NVIDIA A10 and A100 GPUs, configured as per vLLM (Kwon et al., 2023). The setup includes 60 GB of CPU swap space per GPU to facilitate 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 LLMaaS (Large Language Model as a Service) 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, and the priorities are assigned offline, simulating a dynamic and uncontrolled environment where the workload characteristics are highly unpredictable.

Markov This pattern introduces temporal locality into the context switching process. Contexts that have been frequently or recently used are given higher priority. These priorities are also assigned offline, 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 against vLLM. The key metrics we use for evaluation include the p95, p99, and p99.9 *Time to First Token* (TTFT), which measure the latency experienced by the 95th, 99th, and 99.9th percentiles of requests before the first token is generated. Additionally, we consider the p99.9 *Time Between Tokens* (TBT), which assesses the latency between consecutive tokens in the generated responses. These metrics provide a comprehensive understanding of the system's performance, particularly under varying 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 vLLM's 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 task queues in real-time. Upon significant priority changes, the scheduler prioritizes tasks across waiting, running, and swapped queues to align with current priority requirements.

5 Evaluation

In this section, we present the performance evaluation of FastSwitch, highlighting its effectiveness and efficiency compared to the baseline under various conditions.

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 baseline under the QoE metric. 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 (Time to First Token), and p99.9 TBT (Time Between Tokens), 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 attains speedups of 1.4×, 1.5×, 1.4×, and 3.6× for the same metrics.

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-frequency priority changes 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 swap latency, which FastSwitch effectively mitigates through optimized scheduling and resource management.

5.1.3 Summary

As illustrated in Figure 8, the contributions of each subdesign 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.

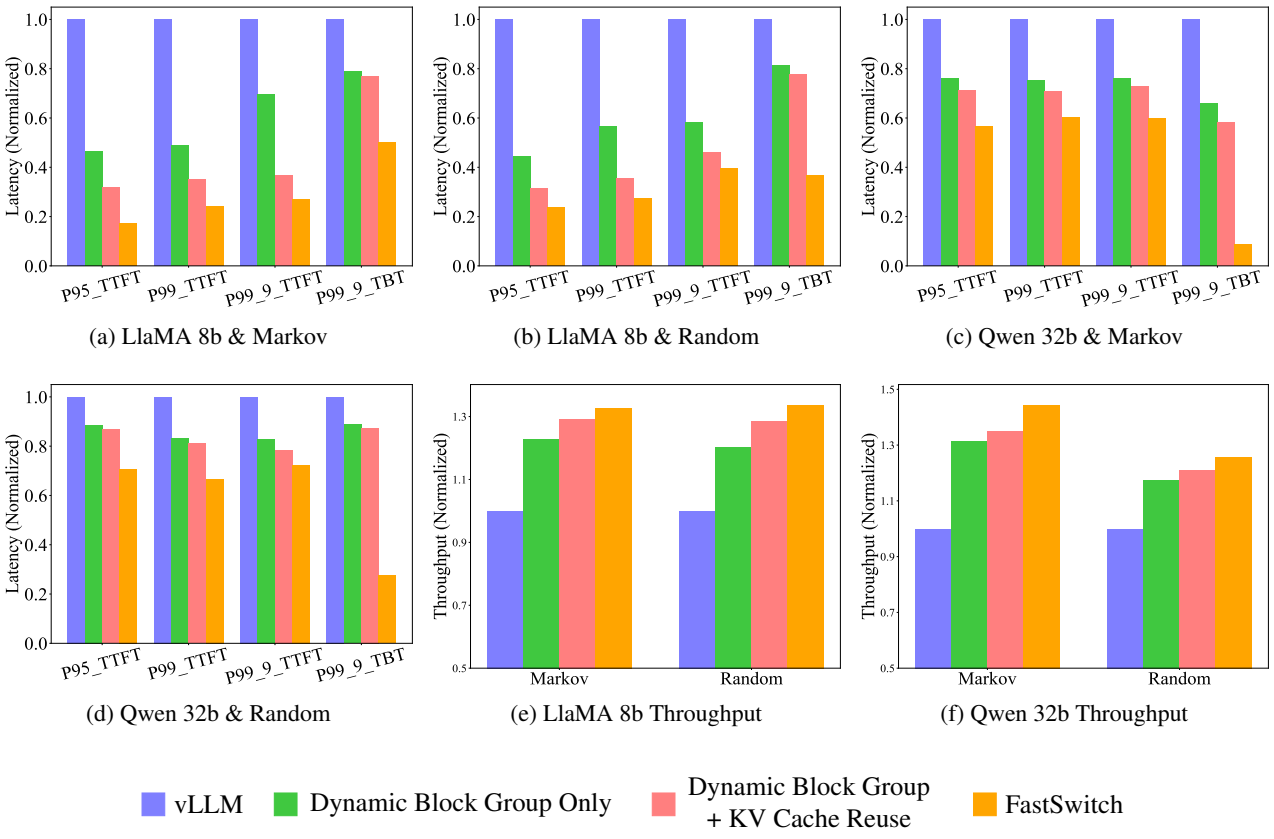

Figure 8. Comparison of TTFT, TBT, and Throughput metrics between FastSwitch and Baseline under different Models and Traces.

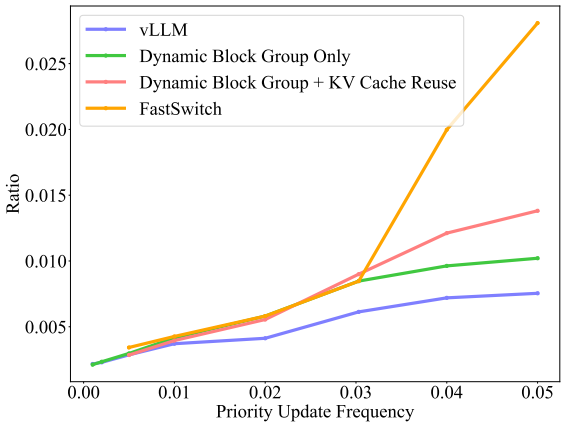

Figure 9. Show the call stack overhead after applying each sub-component or mechanism.

5.2 Call Stack Overhead Analysis

Figure 9 illustrates the call stack overhead as priority update frequency increases. Each component or mechanism of the system introduces optimizations that slightly raise overhead but improve performance. Despite the gradual increase, the overhead contributes to no more than a 1 percent rise in end-to-end time, indicating a minimal impact on overall system efficiency. As the frequency gets higher, the call stack overhead increases accordingly due to the need to resolve more memory conflicts and synchronize more pass iterations with ongoing swap-in requests by the end of the schedule, which results in a certain amount of context switching overhead transferring to the call stack overhead. However, the pure call stack overhead remains within 1 percent.

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 mean 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 memory allocation approach, resulting in simplified swap 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.

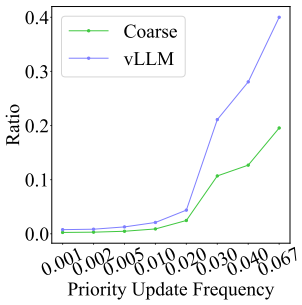

Figure 10. Context switching overhead across priority-update frequencies.

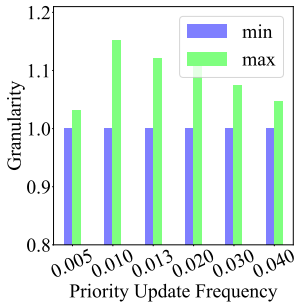

Figure 11. Sensitivity Study of Dynamic Block Group Size.

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 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 swap 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 kvcache for each task, making memory a key factor in swap efficiency.

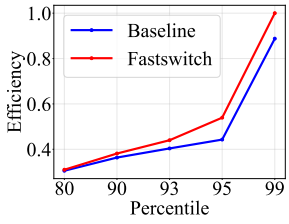

Figure 12. Efficiency Comparison.

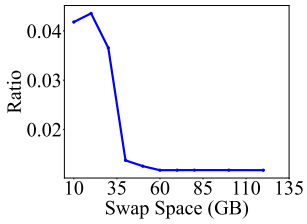

Figure 13. CPU Memory Size Impact

5.3.2 *Multi-threaded Swap Manager*

Token Generation Efficiency The asynchronous execution facilitated by the Multi-threaded Swap Manager improves token generation efficiency. In Figure 12, we compare the token generation efficiency after each priority change between the baseline and FastSwitch. After introducing the Multi-threaded Swap Manager, 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 Multi-threaded 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 Multi-threaded Swap Manager in minimizing the impact of KV cache transfers on overall throughput.

Table 2. 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 2, 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 impact the KV Cache Reuse Mechanism by measuring context switching overhead. As shown in Figure 13, increasing memory allows more KV cache copies to be stored, reducing context switching overhead by enabling greater cache reuse and minimizing redundant data transfers.

Our findings show that larger memory allocations 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 Dis-

tributed Inference Serving system that improves scheduling and resource management in distributed environments.

6.2 KV Cache Management

KV caches (Liu et al., 2024c; Qin et al., 2024; Ge et al., 2023) store 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. These caches 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 cache reuse and intra-program parallelism capabilities found in vLLM and SGLang.

7 Conclusion

We presented FastSwitch, a system that optimizes scheduling in LLM inference through advanced memory management and preemption design. FastSwitch enhances preemptive context switching in LLM frameworks, minimizing performance penalties in TTFT and TBT.

Evaluations show FastSwitch delivers various percentiles of TTFT speedups of **1.4×–5.8×**, a **p99.9 TBT** speedup of up to **11.2×**, and a throughput increase of up to **1.44×**. These gains highlight FastSwitch's ability to ensure fairness and user experience in dynamic, large-scale LLM deployments.

In conclusion, FastSwitch provides an efficient, fair 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.

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.