Oneiros: KV Cache Optimization through Parameter Remapping for Multi-tenant LLM Serving

Ruihao Li[1,*] Shagnik Pal[1,*] Vineeth Narayan Pullu[1] Prasoon Sinha[1]
Jeeho Ryoo[2] Lizy K. John[1] Neeraja J. Yadwadkar[1]
[1]The University of Texas at Austin [2]Fairleigh Dickinson University

Abstract

KV cache accelerates LLM inference by avoiding redundant computation, at the expense of memory. To support larger KV caches, prior work extends GPU memory with CPU memory via CPU-offloading. This involves swapping KV cache between GPU and CPU memory. However, because the cache updates dynamically, such swapping incurs high CPU memory traffic. We make a key observation that model parameters remain constant during runtime, unlike the dynamically updated KV cache. Building on this, we introduce *Oneiros*, which avoids KV cache swapping by remapping, and thereby repurposing, the memory allocated to model parameters for KV cache. This parameter remapping is especially beneficial in multi-tenant environments, where the memory used for the parameters of the inactive models can be more aggressively reclaimed. Exploiting the high CPU-GPU bandwidth offered by the modern hardware, such as the NVIDIA Grace Hopper Superchip, we show that *Oneiros* significantly outperforms state-of-the-art solutions, achieving a reduction of 44.8%-82.5% in tail time-between-token latency, 20.7%-99.3% in tail time-to-first-token latency, and 6.6%-86.7% higher throughput compared to vLLM. Source code of *Oneiros* is available at https://github.com/UT-SysML/Oneiros/.

CCS Concepts

• **Computer systems organization → Cloud computing**; • **Computing methodologies → Machine learning**; • **Software and its engineering → Memory management**.

Keywords

Large language models, Multi-tenant inference serving systems

ACM Reference Format:
Ruihao Li[1,*] Shagnik Pal[1,*] Vineeth Narayan Pullu[1] Prasoon Sinha[1], Jeeho Ryoo[2] Lizy K. John[1] Neeraja J. Yadwadkar[1]. 2025. Oneiros: KV Cache Optimization through Parameter Remapping for Multi-tenant LLM Serving. In *ACM Symposium on Cloud Computing (SoCC '25), November 19–21, 2025, Online, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3772052.3772215

1 Introduction

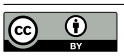
The memory demands of Large Language Models (LLMs) are growing at a much faster rate than GPU memory capacity, making memory a major bottleneck for efficient inference serving [11, 18, 27, 39, 41, 42, 66]. KV cache [27, 49] was introduced as a key mechanism to reduce inference latency: it stores intermediate results from previously generated tokens to avoid recomputing tensor values. To harness the impact of KV cache on the performance of LLM inference serving, numerous memory management techniques are proposed [1, 23, 30, 36, 49, 50], including paging-based mechanisms for runtime KV cache management [27]. However, despite these optimizations, when the KV cache size hits the limit of the GPU memory, one must recompute key-value pairs, increasing serving latency [27] (Figure 1a). The KV-cache bottleneck increases further in multi-tenant environments where multiple LLMs share and contend over the GPU(s) resources [77].

Using CPU memory as an extension of the GPU memory (also called CPU-offloading) is one way to ease the pressure on GPU memory [21, 22, 35, 38, 48, 53, 55, 58, 73]. However, it comes at the cost of increased latency or reduced throughput, depending on the CPU-GPU transfer bandwidth. Prior work [22] explored offloading portions of attention computation to CPUs, allowing KV cache to be generated and stored in CPU memory rather than occupying GPU memory. This allowed the CPU-GPU system to handle larger batch sizes; however, limited CPU performance in attention computation can become a bottleneck, potentially increasing token generation latency by about 10-20% in certain scenarios [22].

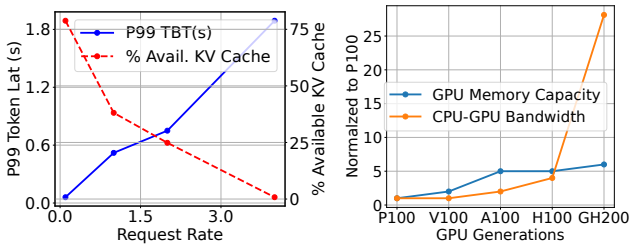
To make it practical, new advancements from the hardware community have dramatically increased the CPU-GPU data transfer bandwidth: the latest Grace Hopper superchip (GH200) [41] introduces a new NVLink-C2C interface between CPU and GPU, offering 7× bandwidth of the H100 [42] and 14× that of the A100 [39] (Figure 1b). This increased CPU-GPU bandwidth opens new opportunities to design low-latency and high-throughput LLM inference serving systems. At the same time, with the availability of higher bandwidth, the limited parallelism of CPUs, rather than the bandwidth, may emerge as the new performance bottleneck (§ 3.1).

Existing work, such as Pie [73], leverages this extra CPU-GPU bandwidth by offloading the KV cache of future layers to CPU memory and swapping back to GPU memory as required, thereby increasing the effective KV cache size (see Figure 2 (left side)). This allows for larger batches or longer sequences without triggering recomputation, thereby reducing inference serving latency and improving serving throughput. However, the frequent updates to the KV cache in GPU memory require synchronization during KV cache swapping, introducing overheads with bidirectional data transfers. As shown in Figure 2 (left side), the KV cache in GPU memory cannot be freed or overwritten (step 2) until the swap to CPU memory (step 1) is complete, since the KV cache is updated after each token generation iteration. We empirically note that this dependency introduces frequent synchronization between CPU

*Equal contribution.



This work is licensed under a Creative Commons Attribution 4.0 International License.
SoCC '25, Online, USA
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2276-9/2025/11
https://doi.org/10.1145/3772052.3772215



(a) Existing LLM systems. (b) GPU Trends.

Figure 1: (a) Existing LLM systems perform recomputation when KV cache is exhausted, leading to a significant increase in latency under high request rates (OPT-13b with ShareGPT). (b) CPU-GPU bandwidth in Nvidia GPUs has increased significantly since GH200, bringing CPU-offloading opportunities.

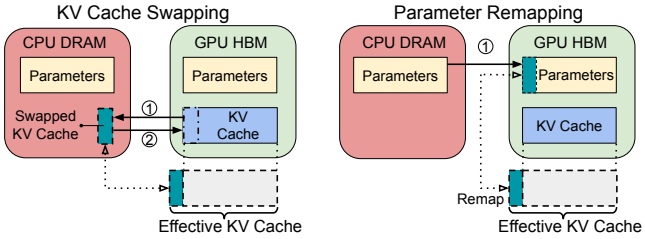


Figure 2: Both KV cache swapping and parameter remapping expand the effective KV cache capacity by utilizing CPU memory as an extension of GPU memory. KV cache swapping incurs overhead due to frequent synchronization, as it requires backing up KV cache to CPU memory after each token generation once swap begins. Instead, parameter remapping is unidirectional with negligible synchronization overhead.

and GPU memory during KV cache swapping, leading to increased serving latency and reduced system throughput (see § 3.2).

Instead, we propose *dynamic parameter remapping*[1], where we simply repurpose a portion of the GPU memory originally allocated for model parameters to support a larger KV cache, enabling the system to accommodate increased KV cache requirements. Unlike KV cache swapping, parameter remapping introduces negligible synchronization overhead, as model parameters remain constant during inference. As shown in Figure 2 (right), it is a non-blocking, unidirectional data transfer that does not require offloading parameters back to the CPU. This immutability simplifies resource management, as the transfer is both deterministic and fixed in size. Notably, parameter remapping is particularly well-suited for multi-tenant LLM serving: it enables efficient and proactive reuse of memory allocated to inactive models[2] by repurposing their parameter storage as KV cache for active models.

However, realizing a practical parameter remapping engine raises several open questions, including when to initiate and stop remapping, which models to target, how many layers to remap, and which specific layers to select. For example, suboptimal layer selection can trigger excessive CPU-GPU data transfers, resulting in a 12.7% reduction in throughput (§ 7.5). Failing to halt remapping in a timely manner can introduce unnecessary pipeline overhead [20, 73], negatively impacting system performance and increasing serving latency by up to 49% (§ 7.6). The complexity of these decisions is further amplified by the dynamic nature of system load and the variability in request sequence lengths across inputs [46, 60, 77]. As a result, these decisions must be made adaptively at runtime, highlighting the need for a sophisticated memory management and data transfer engine within the LLM inference serving system (§ 3.4).

In this paper, we introduce *Oneiros*, a *Dynamic Remapping Engine* that automatically and elastically adjusts KV cache size at *layer* granularity based on *runtime* demands. Such flexibility allows *Oneiros* to be integrated into *any LLM inference serving system with any schedulers* (§ 8). We make the following key contributions:

- We study several GPU memory management optimization strategies to motivate dynamic parameter remapping.
- We introduce *Oneiros*, a *Dynamic Remapping Engine* for multi-tenant LLM inference serving systems that elastically adjusts the KV cache size at runtime by remapping parameter memory as KV cache memory.
- We create a layer eviction algorithm that leverages the circular execution characteristics of LLMs to select which layers to remap. In addition to empirical evaluation, we provide a theoretical proof showing the optimality of our algorithm.
- We integrate *Oneiros* into vLLM to evaluate its effectiveness. Compared to the baseline vLLM, *Oneiros* increases throughput by 7.9%-86.7% and reduces tail latency by 20.7%-99.3%.

2 Background

This section provides background on the state-of-the-art inference serving system (§ 2.1), highlighting that KV cache is the primary bottleneck in both scenarios (§ 2.2).

2.1 LLM Inference Serving

Inference Execution Workflow: LLM inference serving consists of two phases: (a) the *prefill* phase, which processes the full input sequence in a single pass, and (b) the *decode* phase, which generates one output token at a time. Figure 3 illustrates the data placement and execution workflow for a typical LLM inference serving system [27]. The two primary contributors to GPU memory are LLM model parameters and the KV Cache [2, 27, 49, 50]. ① Before the inference serving phase, the model parameters are transferred from CPU memory (DRAM) to GPU memory (HBM) [2, 13, 27, 34, 50]. After loading parameters, ② input tokens are then loaded into GPU memory, where ③ tokens are transferred to GPU compute units (where LLM kernel executes) for processing during the *prefill* phase. During the *decode* phase, the KV cache is generated and expands with the sequence length. Throughout each iteration, the KV cache is transferred between ④ LLM kernels and ⑤ GPU memory. When the maximum sequence length is reached or an end-of-sequence (EOS) token is generated, the output tokens produced by the GPU compute units are first written to GPU memory ⑥, and then transferred back to CPU memory ⑦.

KV Cache: The KV cache stores intermediate results from previously generated tokens during the *decode* phase, eliminating the need to recompute previously generated tensor values. Its size scales linearly with both the batch size and request sequence length.

[1] In this paper, swap refers to bidirectional data transfer to and from CPU/GPU memory, whereas remap denotes unidirectional data transfer from CPU to GPU memory.
[2] Prior work observed that models frequently enter idle periods without incoming requests [77]. We refer to models in such intervals as inactive models.

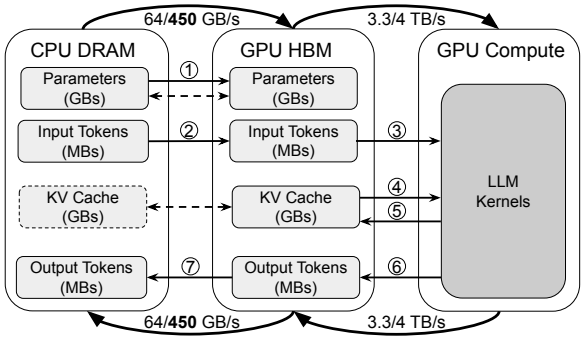


Figure 3: LLM inference serving on GPUs. GH200 features 450 GB/s CPU-GPU bandwidth, while H100 only has 64 GB/s. Swapping KV cache/parameter between CPUs and GPUs during runtime can fully utilize the increased bandwidth.

Performance Metrics: The performance of an LLM inference serving system is typically characterized by two latency metrics – *TTFT* (time-to-first-token) and *TBT* (time-between-tokens) – and by *throughput*. TTFT measures the delay before the first token is generated, indicating system responsiveness, while TBT captures the interval between consecutive tokens, reflecting response smoothness. Throughput quantifies how many requests or tokens the system can process per unit time.

2.2 Memory Bottlenecks in LLM Serving

The memory demands of LLMs have increased at a much faster rate than GPU memory capacity, making memory a critical bottleneck in LLM inference serving. As model sizes grow, they require not only greater computational resources but also significantly more memory to accommodate both model parameters and KV cache [2, 27]. To address this challenge, recent works focused on *efficiently managing KV cache*, as its size grows linearly with the sequence length and batch size, which are unknown prior to runtime [1, 23, 27, 30, 36, 49, 50]. For example, vLLM employs *PagedAttention* [27], which leverages a paging mechanism to manage KV cache pages during runtime. However, when the KV cache exceeds the available GPU memory, recomputation becomes necessary. This scenario is difficult to anticipate, as the sequence length is unknown before runtime and request arrival patterns vary in production environments [46, 59, 60]. This unpredictability also affects multi-tenant LLM inference systems. Regardless of the scheduling policy, systems using *PagedAttention* face the same challenge as single-model serving: runtime KV cache usage is inherently unpredictable. Recomputation can significantly degrade the performance of the affected sequence, especially by increasing the tail latency of the overall serving system. To mitigate this, a mechanism is desired to prevent recomputation when KV cache capacity is exceeded, thereby reducing tail latency in LLM inference serving.

3 Motivation

This section presents three approaches to expanding KV cache capacity, enabled by the significantly higher CPU-GPU bandwidth of modern hardware architectures like GH200.

- Approach 1: offloading computation to CPUs [22], where a portion of the attention operations is offloaded to CPUs, allowing the KV cache to be generated and stored in CPU memory instead of consuming GPU memory. As a result, the combined CPU-GPU system can accommodate larger batch sizes, leading to increased throughput (§ 3.1).
- Approach 2: KV cache swapping [73], which swaps out portions of the KV cache to CPU memory when not immediately needed, freeing up GPU memory for active KV cache usage. This allows the system to accommodate larger batch sizes within the available GPU memory (§ 3.2).
- Approach 3 (proposed): *parameter remapping*, which dynamically repurposes model parameter memory to automatically and elastically expand KV cache capacity at runtime, enabling support for larger batches or longer sequences (§ 3.3).

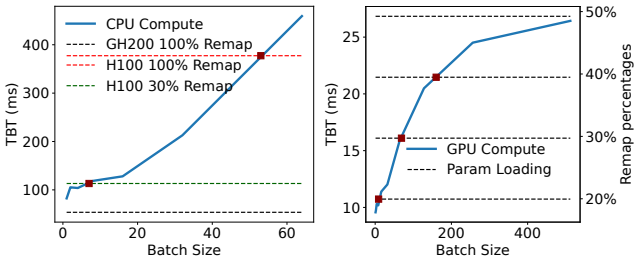
KV cache swapping (Approach 2) and parameter remapping (Approach 3) better utilize the available bandwidth without incurring substantial performance overhead and are more effective than offloading computation to CPUs (Approach 1), which is primarily suited to legacy systems with limited bandwidth. We further compare KV cache swapping (Approach 2) with parameter remapping (Approach 3) and show that parameter remapping has the potential to offer even better performance, particularly in multi-tenant LLM inference scenarios. We conclude with a quantitative analysis of computation and parameter loading times, highlighting challenges and the importance of efficient parameter remapping (§ 3.4).

3.1 Limitations of CPU Offloading

Prior works have demonstrated that offloading part of the attention computation and KV cache states from GPUs to CPUs can effectively increase the batch size and thus inference throughput in LLM inference serving systems [22]. Although offloading computation to CPUs has shown promising performance on earlier hardware platforms, we find it is less efficient on advanced systems such as the GH200 superchip, where limited CPU parallelism, rather than CPU-GPU bandwidth, is the bottleneck. Figure 4a presents the CPU computation time for processing OPT-13b across varying batch sizes serving requests from the ShareGPT dataset [65]. We compare two approaches: (a) offloading computations to CPUs, and (b) continuing GPU computation while remapping parameters to expand KV cache capacity to support large batch sizes. Parameter remapping is preferable when the parameter loading time does not exceed the CPU computation time. In GH200 systems, high CPU-GPU bandwidth but limited CPU parallelism make parameter loading significantly faster than CPU processing, reinforcing the suitability of parameter remapping. In contrast, on an H100 system, offloading computation to CPUs may be viable, even when compared to a low remapping percentage (the fraction of memory used by model parameters that are remapped as added capacity for KV cache), such as 30%. Therefore, on systems with high CPU-GPU bandwidth, remapping parameters offers a more promising solution than offloading computation to CPUs.

3.2 Limitations of Swapping KV Cache

With emerging hardware offering higher CPU-GPU bandwidth, KV cache for non-executing (future) layers can be temporarily swapped into CPU memory and brought back to GPU memory when needed. This approach enables larger batch sizes and enhances overall system performance, as demonstrated by Pie [73].



(a) CPU Compute. (b) GPU Compute.

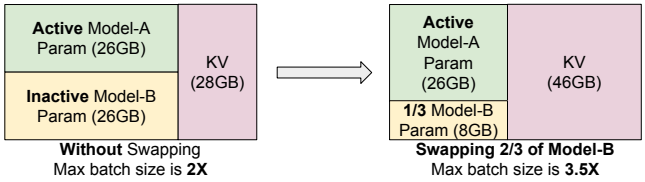
Figure 4: (a) Computation time of different batch sizes when offloading to CPUs. Parameter remapping becomes a more promising approach when CPU-GPU bandwidth is sufficient. (b) GPU computation time fluctuates with batch size based on the load in an online serving system, requiring the parameter remapping percentage to be adjusted dynamically.

However, KV cache swapping presents significant challenges, primarily because the KV cache must be updated after every token generation [2, 27, 76]. This results in frequent synchronization between CPU and GPU memory, introducing overhead that affects serving latency. In addition to synchronization overhead, the bidirectional nature of data transfers can reduce effective CPU-GPU bandwidth, preventing the system from achieving an ideal pipeline of data transfer and computation. Our measurements of CPU host memory (LPDDR5X[3]) bandwidth show that unidirectional (read-only of CPU host memory) CPU-to-GPU transfers can reach ~427 GB/s. However, when the read-to-write ratio is 1:1, bandwidth drops to ~366 GB/s, resulting in a ~15% reduction. This bandwidth degradation can further impact serving throughput.

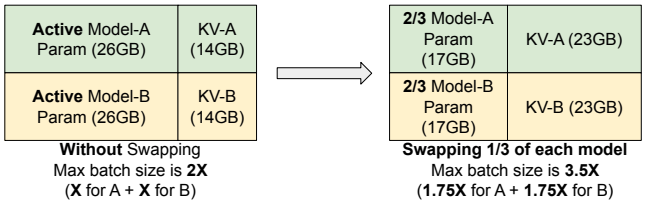
3.3 Our Proposal: Parameter Remapping

We introduce Parameter Remapping, where, instead of swapping KV cache, we propose to dynamically remap the space used for model parameters to automatically and elastically expand the KV cache size at runtime. Dynamic parameter remapping reclaims GPU memory by repurposing a portion of the model parameters for KV cache usage during runtime, enabling online inference serving systems to handle peak loads when GPU memory becomes insufficient for KV cache. As shown in the two examples in Figure 5, deploying two OPT-13b models on an H100 GPU using either temporal (Figure 5a) or spatial (Figure 5b) sharing one GPU consumes approximately 52GB of the available 80GB memory for model parameters [27], leaving only about 28GB for KV cache. By remapping one-third of the model parameters as KV Cache – either 2/3 from the inactive model in temporal sharing or 1/3 from each model in spatial sharing – the maximum batch size can be increased by up to 1.75×, resulting in a corresponding 1.75× improvement in system throughput. Unlike KV cache swapping, parameter remapping involves non-blocking, unidirectional data transfers and does not require backing up model parameters from GPU to CPU, as parameters remain unchanged during inference. As a result, synchronization overhead is significantly reduced. Additionally, parameter remapping is particularly effective in multi-tenant LLM inference scenarios, such

[3]The trend observed in our measurements also holds for other memory technologies (e.g., DDR4, DDR5) beyond LPDDR5X [12].



(a) Temporal sharing (remap 2/3 param of inactive Model-B).



(b) Spatial sharing (remap 1/3 param of each model).

Figure 5: Remapping parameters for KV cache allows for larger batches when deploying two OPT-13b models on H100, in both (a) temporal sharing and (b) spatial sharing scenarios.

as multi-agent workflows [5, 17, 26, 31, 63] and production deployments where many models remain idle for long periods of time [77]. In such settings, memory allocated to inactive models can be reclaimed for active ones, a flexibility that KV cache swapping lacks, as inactive models may have no KV cache to be swapped.

3.4 Challenges in Parameter Remapping

While parameter remapping can improve performance, it introduces potential overhead and requires careful design. To avoid impacting inference latency and throughput, parameter loading must ideally overlap with GPU computation, ensuring that the parameters of each layer are fully loaded before its execution begins. For example, while the GPU executes layer 1, layer 2's parameters can be concurrently loaded from CPU to GPU memory. The ideal case is when layer 2 finishes loading before layer 1 finishes execution, allowing GPU memory to hold only two layers at a time [54, 58], unlike approaches that preload the entire model into GPU memory. Despite GH200's higher CPU-GPU bandwidth, it remains insufficient for full on-demand remapping, since layer execution outpaces parameter transfer. As a result, only a fraction of the parameters, defined by the *remapping percentage*, can be dynamically transferred at runtime, while the rest must be preloaded into GPU memory before the inference serving starts. In this section, we examine the challenges involved in making these design decisions.

When to trigger remapping? While an ideal pipeline can fully overlap GPU computation with parameter loading from CPU to GPU memory, the parameter remapping engine needs to initiate parameter remapping only when necessary, as even a perfectly pipelined approach introduces inherent overhead [20]. Likewise, it is important to halt parameter remapping once it is no longer needed. Delaying this decision can increase the latency of the serving system by up to 49% (§ 7.6). These decisions require dedicated mechanisms and careful design for online monitoring of the LLM inference serving system.

Which models to remap? In multi-tenant LLM inference serving systems, not all models are active simultaneously [47, 77], which presents an opportunity for the parameter remapping engine to prioritize remapping parameters of inactive models. However, this

approach requires careful design, as overly aggressive offloading may lead to starvation – especially given the unpredictable nature of request arrival patterns in online serving environments [60]. An inefficient design is likely to prevent certain models from responding promptly; for example, a suboptimal model selection can lead to a 22.0% increase in serving tail latency (§ 7.6).

How many layers to remap? Determining the optimal remapping percentage is challenging due to the dynamic nature of request arrival rates in real-world online inference serving [46, 59, 60]. The fluctuations in request arrival rates can affect the GPU computation time, which in turn affects the percentage of parameters that can be transferred without performance degradation. We illustrate the need for dynamic adjustment of the remapping percentage using an example of serving OPT-13b with varying batch sizes on the ShareGPT dataset [65]. Figure 4b shows how the GPU computation time for generating one token in the *decode* phase varies with batch size, while parameter load time – determined solely by the remapping percentage – remains constant. The intersection points of these curves represent configurations where parameter loading is fully overlapped with computation, maximizing efficiency. As shown, the optimal remapping percentage varies with changes in the runtime batch size, reflecting shifts in system load. Using a static remapping percentage may result in suboptimal resource use: underutilizing GPU memory at off-peak load and causing compute stalls at peak load. For example, a 30% remapping percentage tuned for average batch size in ShareGPT results in CPU-GPU bandwidth becoming a bottleneck during off-peak workloads. Therefore, the remapping percentage must be dynamically adapted at runtime.

Which layers to target? Once the remapping percentage is set, the next design challenge is selecting specific layers to remap. This remains non-trivial, as *Oneiros* must decide how to allocate limited GPU memory across all layers – deciding which layers remain permanently in GPU memory and which are transferred on the fly. For an efficient pipeline, a layer selection strategy is needed to guide the layer-sharing mechanism in *Oneiros*. A suboptimal layer selection can lead to reduced serving throughput (by 12.7%, § 7.5).

4 *Oneiros* Overview

We introduce *Oneiros*, a *Dynamic Remapping Engine* for multi-tenant LLM inference serving systems that elastically adjusts the KV cache size by remapping parameter memory as KV cache memory. Figure 6 shows the *Oneiros* architecture, and depicts how *Oneiros* can be integrated into existing LLM inference serving systems.

4.1 Architecture of *Oneiros*

Oneiros is designed to have three key components, marked in green color in Figure 6.

Metadata Store: The *Metadata Store* maintains the information about the models and their memory utilization. Model information includes the currently scheduled models (referred to as active models) and idle models currently not receiving requests (referred to as inactive models). Memory utilization information captures KV cache usage, with a focus on the amount of available free memory.

Remapping Controller: The *Remapping Controller* dynamically repurposes GPU memory allocated to inactive model parameters for use as KV cache by active models. When KV cache demand exceeds available GPU memory, the *Remapping Controller* reclaims

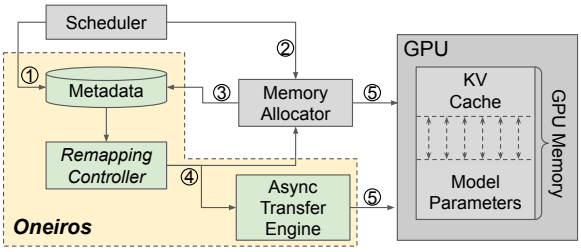


Figure 6: Overview of *Oneiros*.

a portion of the parameter memory to expand KV cache capacity. Conversely, when KV cache usage subsides, the reclaimed memory is remapped back for parameter storage.

Async Transfer Engine: The *Async Transfer Engine* manages parameter loading and synchronizes CPU-GPU data transfers with GPU computation. When the *Remapping Controller* initiates parameter remapping, a portion of GPU memory originally allocated for model parameters is repurposed for KV cache usage. If the reclaimed parameters are later required for computation, the *Async Transfer Engine* fetches them back from CPU memory to GPU memory. By leveraging the deterministic execution order of layers within each *decode* iteration, the *Async Transfer Engine* overlaps CPU-GPU data transfers with ongoing GPU computation.

4.2 Workflow

Figure 6 depicts the workflow of the interaction between various components to achieve the goal of dynamic parameter remapping in *Oneiros*. The *Scheduler* selects which models to serve, referred to as active models. Once active models are chosen, the *Memory Allocator* updates the parameter and KV cache memory usage accordingly. Using both model information and KV cache utilization data, the *Remapping Controller* determines whether parameter memory needs to be remapped to expand KV cache capacity. If remapping is required, the *Remapping Controller* updates memory usage records in the *Memory Allocator* and triggers the *Async Transfer Engine* to initiate data transfer. The GPU then executes the LLM inference tasks. With support from *Oneiros*, a portion of parameter memory in GPUs can be dynamically reclaimed at runtime, allowing the system to accommodate larger batches or longer sequences.

5 Remapping Controller

Oneiros enables dynamic, runtime management of GPU memory through its *Remapping Controller*. Figure 7 illustrates an example execution workflow of the *Remapping Controller* across five phases, each representing a snapshot of model activity within an LLM serving system. Depending on runtime conditions, the *Remapping Controller* decides either to initiate remapping, continue remapping additional layers from the active or inactive models, or halt remapping altogether (reclaimed KV cache memory restored for parameter storage). In this section, we describe how *Remapping Controller* answers the key questions, including when to remap (§ 5.1), which models to remap (§ 5.2), how many (§ 5.3), and which layers to remap (§ 5.4). Putting it all together, we then present an algorithm that can seamlessly integrate *Oneiros* into existing LLM inference serving frameworks, enabling dynamic remapping of model parameter memory to automatically and elastically expand KV cache capacity at runtime (§ 5.5).

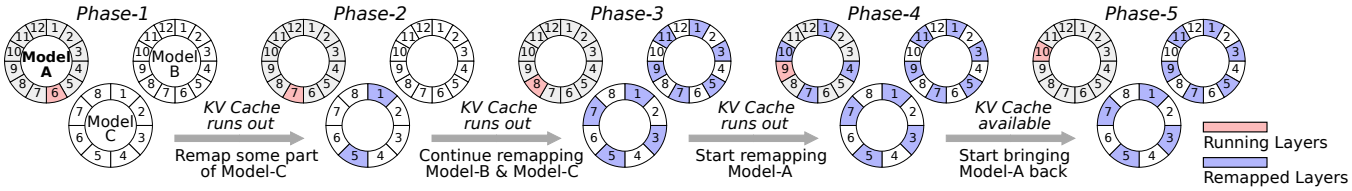


Figure 7: *Remapping Controller* decides which model and layer to remap based on runtime KV cache usage (Model-A is active).

5.1 When to Remap?

The *Remapping Controller* initiates parameter remapping when the KV cache capacity is exhausted. While unidirectional parameter remapping incurs lower overhead than bidirectional KV cache swapping, *Oneiros* activates it only when necessary. As such, accurately determining both the initiation and termination points of remapping is critical. Given the dynamic nature of real-world production workloads [7, 15, 46, 57, 59, 60], the *Remapping Controller* enables remapping during peak request periods to accommodate larger batch sizes. During non-peak periods, when KV cache space is sufficient, remapping is disabled to avoid the overhead of reloading previously evicted parameters into GPU memory. The reclaimed KV cache memory is then restored for parameter usage.

5.2 Which Models to Remap?

This section explains how the *Remapping Controller* determines which models to remap. We begin by reviewing existing GPU sharing techniques used in multi-tenant inference serving systems, as the *Remapping Controller* adapts its model selection strategy based on the specific sharing mechanism employed. We then present a detailed explanation of our model selection strategy.

GPU sharing background: Given the memory capacity of emerging GPUs, enabling GPU sharing is essential to fully utilize their processing capabilities. Prior works have examined both temporal [4, 13, 14, 47, 71] (example in Figure 5a) and spatial [6, 9, 16, 28, 56, 61, 70, 72] (example in Figure 5b) sharing strategies for efficient machine learning inference serving on GPUs. Temporal sharing allocates the entire GPU to one model at a time, allowing for larger batches per model. This approach is especially useful in multi-agent workflows – where the output of one model becomes the input to another [5, 17, 26, 31, 63] – and in cloud production environments, where models often experience idle periods with no incoming requests [77]. In contrast, spatial sharing divides GPU resources among multiple models simultaneously, allowing for faster response times at the expense of smaller batch sizes per model [10, 28]. *Each GPU resource sharing strategy is tailored to specific use cases and can be selectively employed by multi-tenant LLM inference schedulers. The Remapping Controller in Oneiros is designed to be compatible with any scheduling policy, although the design of the scheduler itself is beyond the scope of this study.*

Model selection strategy: We now present the model selection strategy to determine which models to remap parameter layers from. The *Remapping Controller* prioritizes remapping parameters from inactive models with the lowest priority, as determined by the scheduling policy, rather than from active ones. Under temporal GPU resource sharing, models scheduled for future execution are considered inactive. In contrast, under spatial sharing, all models are treated as active, requiring each to remap parameters. The number of layers to remap and their selection are determined using the strategies described in § 5.3 and § 5.4. If the scheduler does not specify model priorities or all models are active, the *Remapping Controller* defaults to a round-robin policy, treating all models equally, thereby ensuring fairness across models. In this case, it applies a Most Recently Used (MRU) strategy, remapping parameters from the most recently activated model to defer parameter transfer costs as far into the future as possible. To avoid any starvation, *Oneiros* also caps the maximum number of layers remapped based on a threshold derived empirically through runtime profiling (more in § 5.3), and no model is ever remapped in its entirety. This strategy is especially effective when future request patterns are unpredictable or when model priorities are unspecified.

Example: Phase 2 of Figure 7 illustrates an example of the model selection policy. In this scenario, Model-A is active, while Model-B and Model-C are inactive, with the scheduler assigning the lowest priority to Model-C. The *Remapping Controller* first evicts and remaps parameters from Model-C. To avoid starving inactive models, a maximum remapping threshold is enforced (see § 5.3), ensuring that enough parameters remain in GPU memory to support cold-start loading during the *prefill* phase when a model becomes active. After reaching the remapping limit for Model-C, the *Remapping Controller* continues with Model-B. Only once all inactive models have reached their remapping thresholds does the *Remapping Controller* begin remapping parameters from the active Model-A.

5.3 How Many Layers to Remap?

To hide parameter transfer latency by overlapping it with GPU computation, the transfer from CPU to GPU must finish before the computation completes; otherwise, it becomes a performance bottleneck. Since both model parameter sizes and CPU-GPU bandwidth are known ahead of runtime, the transfer time per layer can be profiled offline and treated as prior knowledge by the online inference system. As a result, the *Remapping Controller* only needs to monitor GPU computation time, denoted as $T_{Compute}$, at runtime to determine whether additional parameters can be remapped. For an active model, $T_{Compute}$ refers to the time required to generate the current token during the *decode* phase. For an inactive model, it represents the duration of the *prefill* phase, which is more compute-intensive and typically longer, as no tokens have been generated yet [2, 46]. Let T_T denote the time required to transfer a single layer of parameters. To prevent parameter loading from becoming a new bottleneck, the number of remapped layers N must satisfy the constraint $T_T \cdot N \leq T_{Compute}$.

The longer $T_{Compute}$ in the *prefill* phase allows more layers in inactive models to be remapped compared to when the model is active. Similar to the decode phase, the *prefill* phase executes one layer at a time rather than all at once. If all the parameters are not present in GPU memory because of the parameter remapping triggered by *Oneiros*, the *Remapping Controller* transfers the parameters

for the later layers from CPU memory back into the GPU memory while initial layers are being executed. Thus, the parameters are ready in GPU memory when needed, as the *prefill* phase makes progress layer-by-layer. The *prefill* phase is compute-bound, and takes additional compute time that provides an opportunity for *Oneiros* to hide the latency of loading remapped parameters back to GPU memory.

5.4 Which Layers to Target?

This section details the layer selection strategy in the *Remapping Controller*. We begin by describing the strategy itself, followed by an example to illustrate its behavior. Finally, we provide a theoretical proof to demonstrate that the strategy we used is optimal.

Layer selection strategy: Understanding the layer selection strategy requires recognizing the circular execution pattern of layers in LLM inference, as shown in Figure 7. The auto-regressive nature of LLM inference makes each of the subsequent tokens execute the same layers again. Thus, LLM inference can be viewed as a circular execution of layers, where the execution of the last layer for the current token is succeeded by the execution of the first layer for the next token. The *Remapping Controller* uses a uniform-interval layer selection strategy, in which evenly spaced layers are chosen to share a common GPU memory region allocated for their parameters. These layers alternate access to the shared region – at any given moment, some layer parameters reside in GPU memory, while others remain in CPU memory. When a layer not currently in GPU memory is needed, its parameters are loaded into the shared region, replacing those of the previously loaded layer.

Example: To provide a more intuitive explanation, we use the example of Model-C with 8 layers (Phase 2 of Figure 7). The *Remapping Controller* employs a uniform-interval layer selection strategy. For instance, when remapping the parameters of a single layer, Layers 1 and 5 are selected to share a common GPU memory region (the other 6 layers remain in GPU memory permanently), alternating between CPU and GPU memory. The uniform-interval layer selection strategy is designed to balance serving latency across successive token generations. In contrast, ignoring the circular nature of LLM inference – by treating each token generation as an independent execution – may lead to suboptimal choices, such as selecting Layer 1 and Layer 8. Although this configuration allows sufficient time to load the parameters of Layer 8 after executing Layer 1, it does not provide enough time to reload the parameters of Layer 1 before it is needed again. This demonstrates the benefit of uniform-interval selection in circular execution, which ensures that each remapped layer can be restored to GPU memory in time for reuse.

Theoretical proof: Consider an n-layer model in which the *Remapping Controller* remaps the parameter memory of α layers for KV cache. Consequently, all n layers must be scheduled within the GPU memory footprint of only $n - \alpha$ layers. Suppose m layers are transferred between CPU and GPU during each token generation. We note $L_1, L_2, \ldots, L_m$ as the m layers (remapped layers in Figure 7) that are transferred from CPU to GPU. We define T_T as the time taken to load parameters of a single layer from CPU to GPU memory, and T_c as the time spent on computation for a single layer on GPU. *Our layer selection policy aims to satisfy two key objectives: (1) ensure that parameter transfer latency does not become a performance bottleneck; and (2) minimize CPU-GPU data transfer traffic.*

To satisfy objective (1): the CPU-to-GPU parameter loading must be fully overlapped with GPU computation. During each token generation, the *Remapping Controller* transfers m layers from CPU to GPU memory. The goal is to select m layers such that the computation between their executions is sufficient to hide the associated transfer latency. For example, after layer L_1 completes execution, the *Remapping Controller* begins transferring parameters of layer L_2. The cumulative computation time of the layers executed between L_1 and L_2 must be long enough to mask the transfer time of L_2. Similarly, each subsequent transfer must also meet the condition to hide its transfer time, continuing until the parameters for layer L_1 (for the next token) are fully transferred. We formally present this constraint as:

$$T_T \leq \sum_{i=L_1}^{L_2} T_c \ \cdots, \ T_T \leq \sum_{i=L_{m-1}}^{L_m} T_c, \ T_T \leq \sum_{i=L_m}^{L_1} T_c \tag{1}$$

Consider that there are k_1 layers between L_1 and L_2[4], and k_m layers between L_m and L_1 (for the next token), we have:

$$\sum_{i=L_1}^{L_2} T_c = k_1 \times T_c, \ \cdots, \ \sum_{i=L_m}^{L_1} T_c = k_m \times T_c \tag{2}$$

Leveraging Equations 1 and 2, we conclude that:

$$T_T \leq \min(k_1 \times T_c, \ \cdots, \ k_{m-1} \times T_c, \ k_m \times T_c) \tag{3}$$

The Right-Hand Side (RHS) of Equation 3 denotes the threshold beyond which transfer time becomes a performance bottleneck. To maximize the RHS under the constraint $k_1 + k_2 + \cdots + k_m = n$, the optimal solution is achieved when $k_1 = k_2 = \cdots = k_m$. In other words, *within the circular layer execution model, the selected m layers must be evenly spaced, ensuring uniform intervals between consecutive remapped layers.* This aligns with the geometric principle that, for n points on a circle, the configuration maximizing the minimum pairwise distance is achieved by placing them at equal angular intervals.

To satisfy objective (2): we aim to find the smallest valid integer value of m. We begin with $m = \alpha$, meaning that only α layers are loaded from CPU memory during each token generation. However, this configuration is infeasible because loading a remapped layer into GPU memory requires evicting a resident layer, which will be needed in the next token generation due to the circular reuse pattern of all layers. As a result, $m = \alpha$ is insufficient, and the minimum viable value becomes $m = \alpha + \beta$ where $\beta \geq 1$. In this configuration, $n - (\alpha + \beta)$ layers remain permanently in the GPU memory, while the remaining $\alpha + \beta$ layers share a GPU memory region sized for β layers.

We start with $\beta = 1$. To satisfy design objective (1), we have Equation 3 to make the data transfer time shorter than the computation time. However, the prerequisite of Equation 3 is that parameter transfers from CPU to GPU can be initiated at any time, overlooking the data dependencies between layers: the $\alpha + 1$ layers share a single GPU memory slot, meaning that the transfer of a layer cannot begin until its computation has completed. Accounting for

[4]In this paper, we assume all layers are identical. For emerging models with heterogeneity across layers [79], these formulas can be easily adapted, e.g., T_T can be written as $T_{T,i}$, where i denotes each specific layer, and solved using integer linear programming.

this constraint, the time budget becomes more restrictive:

$$T_T \times (\alpha + 1) \leq T_c \times (n - \alpha - 1) \quad (4)$$

To relax this constraint, one possible solution is to adopt a double-buffering approach, increasing the number of layers transferred on the fly from $\alpha + 1$ to $\alpha + 2$. With two GPU memory slots, these $\alpha + 2$ layers can be transferred and computed in a staggered manner, allowing the transfer of the next layer to begin without waiting for the execution of the current layer to complete. This results in a less restrictive timing condition:

$$T_T \times (\alpha + 2) \leq T_c \times n \quad (5)$$

When α is small and n is large, setting $m = \alpha + 1$ is generally preferred as it minimizes CPU-GPU transfer. In contrast, $m = \alpha + 2$ is more suitable for larger models with more remapped parameters. For instance, based on Equations 4 and 5, when $\alpha \geq 9$ for a model with 40 layers ($n = 40$), choosing $m = \alpha + 2$ can provide better performance (§ 7.5).

5.5 Putting All Together

Algorithm 1 outlines the complete execution workflow of the *Remapping Controller*. Consistent with existing LLM inference serving schedulers, the *Remapping Controller* makes parameter remapping decisions at a per-token granularity. At each iteration step, it first evaluates the runtime status of the *Memory Allocator*, to determine whether remapping needs to be triggered (Line 3). Then, it marks the GPU memory previously occupied by parameters as available for KV cache. The *Remapping Controller* then updates the list of model layers whose parameters will be remapped between CPU and GPU memory (Line 4). As illustrated in the remapping() function of Algorithm 1, the *Remapping Controller* continuously tracks the current remapping percentage for each model (Lines 18-23) and prioritizes remapping parameters from the model (Line 16) with the lowest priority decided by the scheduler. Meanwhile, when the KV cache space is freed after requests finish, the *Remapping Controller* updates the remapping percentage based on the available memory information from the memory allocator (Lines 7-12). Finally, the *Remapping Controller* updates parameter remapping information (remapped models and layers) to the GPU LLM Kernel, which executes the current iteration accordingly (Line 13).

6 Implementation

We integrate *Oneiros* into vLLM [27], a state-of-the-art LLM inference-serving framework. We add ~3,000 lines of Python code and ~300 lines of C++/CUDA code to enable support for parameter remapping engines and dynamic KV cache allocation.

Remapping Controller and **Async Transfer Engine:** During inference, the *Remapping Controller* coordinates with the *Transfer Engine* to specify which layers from which models are to be remapped. If remap is disabled, *Oneiros* follows the standard execution flow unchanged. When remap is enabled, the *Transfer Engine* initiates an asynchronous transfer of the parameters of the target layer, overwriting[5] the GPU tensor memory previously used by layers that have completed execution. To ensure correctness, the GPU performs memory barrier synchronization checks before launching kernels that depend on remapped parameters.

[5]State-of-the-art LLM inference frameworks, such as vllm [27], keep a complete copy of all model parameters in CPU memory before transferring to GPU memory.

Algorithm 1: *Remapping Controller* Workflow

```
1  Function Remapping_Controller():
2      foreach step() ∈ LLM Inference Serving Scheduler do
3          if Running out of KV Cache blocks then
4              remapped_layer_list = remapping();
5              enable_remap = True;
6          else
7              foreach Tensor ∈ Added KV Cache do
8                  if Tensor is empty then
9                      remapped_layer_list = remapping();
10             end
11             if All Tensors are empty then
12                 enable_remap = False;
13         GPU LLM Kernel(enable_remap, remapped_layer_list);
14     end
15 Function Remapping():
16     model = pop(inactive_model_list);
17     if model is not NULL then
18         model.remapped_layers++;
19         if model.remapped_layers == model.layers then
20             inactive_model_list.remove(model);
21         sort(inactive_model_list);
22     else
23         remapping parameters of active models;
```

KV Cache Allocation: Most state-of-the-art LLM frameworks, such as vLLM [27], support dynamic management of KV cache memory space. However, the physical memory for KV cache still needs to be statically allocated before serving starts. During parameter remapping, a portion of the parameter memory must be dynamically reconfigured as KV cache memory, requiring support for runtime dynamic KV cache allocation and deallocation. To enable dynamic allocation and remapping of GPU memory, we adopt the vAttention design [50], which reserves sufficient virtual memory space for the KV cache and maps physical memory pages on demand. Once parameter tensors are released, the KV cache engine can immediately reuse the freed physical memory. This approach also allows GPU physical KV cache memory to be efficiently shared across multiple models.

7 Evaluation

We comprehensively evaluate the effectiveness of *Oneiros* by answering the questions below:

- How effective is *Oneiros* in reducing tail latency and improving throughput under temporal GPU sharing? (§ 7.2)
- Is *Oneiros* still effective in enhancing performance under spatial GPU resource sharing? (§ 7.3)
- How effective is the parameter remapping mechanism compared with KV cache swapping? (§ 7.4)
- How effective is the layer selection strategy used? (§ 7.5)
- How important is the decision of when to remap and how many layers to remap? (§ 7.6)

Some highlights of our results include:

- *Oneiros* significantly improves multi-tenant LLM inference with temporal GPU sharing, reducing tail TBT latency by 44.8%-82.5%, tail TTFT latency by 74.8%-99.3%, and achieving 39.9%-86.7% higher throughput compared to vLLM.
- In addition to temporal sharing of GPU resources, *Oneiros* also supports spatial sharing. *Oneiros* achieves 20.7%-65.5% reduction in tail latency.

Table 1: Evaluated model combinations and their corresponding GPU memory reservations (% of total GPU memory).

C1	OPT-13b (35%), Lllma-2-13b (35%), Lllma-3-8b (20%)
C2	OPT-30b (65%), OPT-6.7b (15%)

- Compared to KV cache swapping, *Oneiros* achieves 47.1% higher throughput. Additionally, it supports multi-tenant LLM inference serving, providing greater flexibility.

7.1 Experimental Setup

Platforms: We integrate *Oneiros* into vLLM v0.7.3 [27], using CUDA 12.4, PyTorch 2.5.1, and Linux kernel 5.14. All experiments were conducted on the NVIDIA GH200 Grace Hopper Superchip, which features an H200 GPU with 96GB of HBM3 memory and a CPU host with 72 Arm Neoverse V2 cores and 224GB of LPDDR5X memory, connected via a 900 GB/s NVLink interconnect [41].
Models: We evaluated the following models: OPT [80] with 13b and 30b parameters; Llama-2 [67] with 13b parameters; and Llama-3 [11] with 8b parameters. For multi-tenant serving, we evaluate two model combinations *C1* and *C2* in Table 1.
Traces: We evaluate *Oneiros* using the Azure coding LLM inference traces that capture real-world query arrival patterns characterized by bursty query patterns [46, 60]. Following prior work [3, 14, 34, 37], we scale these traces to different query arrival rates while preserving their original characteristics, such as burstiness.
Datasets: We use the ShareGPT [65] and Alpaca [64] datasets, both of which contain input-output pairs from real LLM services. To simulate realistic multi-tenant scenarios, we additionally generate requests of varying lengths (synthetic datasets).
Baselines: We compare *Oneiros* against two baselines: vLLM [27] and Pie [73]. vLLM is a state-of-the-art LLM inference serving system widely used in both academia and industry. Pie is the first to explore using the NVIDIA Grace Superchip for KV cache swapping.
Metrics: Our evaluation focuses on two key metrics: (a) tail (P99) latency, which is critical for managing congestion in online systems [8, 19, 24, 51, 81]; we consider both TBT and TTFT latency; and (b) throughput (tokens per second), which quantifies the number of tokens a system can handle within a specific time frame.

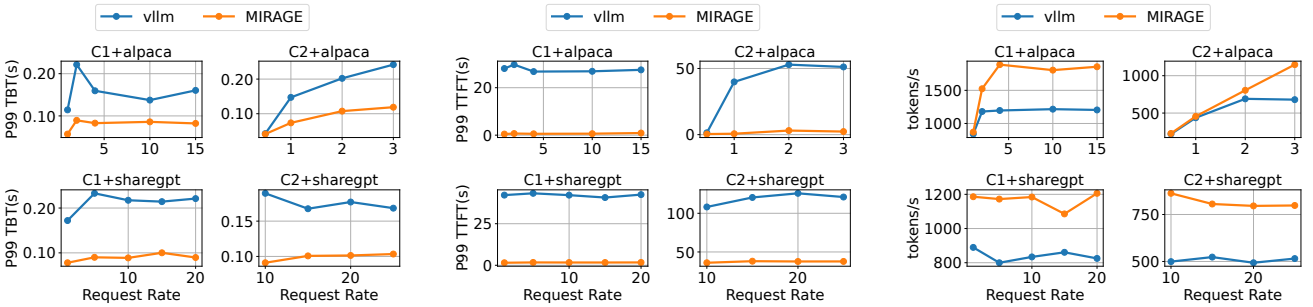
7.2 *Oneiros* Supports Temporal GPU Sharing

We first evaluate *Oneiros* under temporal GPU resource sharing, which is well-suited for multi-agent workflows and scenarios with frequent model idle periods.
Tail Latency: We first evaluate *Oneiros* by comparing tail TBT and TTFT latencies. Figures 8a and 8b present the P99 TBT and TTFT latency comparisons between *Oneiros* and vLLM. Across all scenarios, *Oneiros* consistently reduces tail latency. On average, when serving *C1*, *Oneiros* achieves an average reduction of 54.4% in P99 TBT latency and 96.7% in P99 TTFT latency across both the Alpaca and ShareGPT datasets. Similarly, when serving *C2*, *Oneiros* achieves an average reduction of 44.8% in P99 TBT latency and 74.8% in P99 TTFT latency on the same datasets.

Without *Oneiros*, peak-load conditions that exhaust available KV cache typically cause existing serving systems [27] to pause the *decode* process for all active requests, evict one, and recompute its KV cache to free space. *Oneiros* eliminates the need for recomputation, maintaining low tail latency, thereby ensuring stable performance for online inference serving.
Throughput: By using the *Async Transfer Engine*, *Oneiros* overlaps parameter transfer with GPU computation. This not only reduces tail latency but also sustains high serving throughput. Across all evaluated scenarios shown in Figure 8c, *Oneiros* consistently increases throughput. On average, when serving *C1*, it achieves a 39.9% improvement in throughput across both the Alpaca and ShareGPT datasets. Similarly, when serving *C2*, *Oneiros* delivers an average throughput gain of 45.3% on the above datasets.
Varied Arrival Rates: In real-world multi-tenant scenarios, different models or tasks often experience varying arrival rates [34, 56, 62, 77]. *Oneiros* is designed to handle this variability effectively. To demonstrate its effectiveness, we evaluate *Oneiros* by serving *C2*, with different arrival rates across different models. As shown in Figure 9, *Oneiros* consistently improves performance across all tested scenarios of varied input arrival rates, achieving an average of 86.7% increase in throughput and 82.5% and 99.3% reduction in TBT and TTFT tail latencies, respectively.
Varied Inputs: To emulate realistic multi-tenant scenarios – where different models handle distinct tasks within a multi-agent flow and often operate on separate input datasets [31, 33, 62, 63] – we construct arrival traces for two models (evaluated on *C2*) using two request-type combinations: (a) synthetic long requests (average of 1734 tokens) paired with synthetic short requests (average of 634 tokens); and (b) synthetic short requests paired with synthetic long requests. As shown in Figure 10, *Oneiros* consistently improves performance across all evaluated scenarios, achieving an average throughput increase of 65.6% and reductions of 57.1% and 98.1% in P99 TBT and TTFT latencies, respectively.
Which models to remap? We evaluate *Remapping Controller*'s model selection strategy (§ 5.2) under temporal GPU sharing. This strategy aligns with the multi-tenant scheduling policy that decides model execution order. In the absence of explicit model priorities, *Oneiros* uses a default round-robin scheduling policy with a Most Recently Used (MRU) eviction strategy. To assess its effectiveness, we compare MRU against a Least Recently Used (LRU) policy, which remaps the parameters of the model most likely to be served next. Figure 11 presents performance results for the *C1* model combination under round-robin execution. With the MRU policy, *Oneiros* achieves up to a 22.0% reduction in tail latency compared to the LRU policy. MRU consistently outperforms LRU by deferring transfer costs, remapping parameters of models that are less likely to be reused in the near future.

7.3 *Oneiros* Supports Spatial GPU Sharing

In addition to supporting temporal GPU sharing, *Oneiros* also performs effectively under spatial sharing for online inference. In this setup, all models remain active and maintain separate KV cache spaces. To assess the effectiveness of *Oneiros*, we evaluate the model combination *C1* on both Alpaca and ShareGPT datasets using Azure coding traces. We evaluate the performance of *Oneiros* under two representative spatial sharing methods on NVIDIA-based GPU systems: (1) *Non-strict physical isolation*, as seen in Multi-Process Service (MPS) [44], enables concurrent execution by allowing multiple



(a) *Oneiros* reduces tail TBT latency. (b) *Oneiros* reduces tail TTFT latency. (c) *Oneiros* improves throughput.

Figure 8: *Oneiros* reduces latency and improves throughput compared with vllm on both C1 and C2.

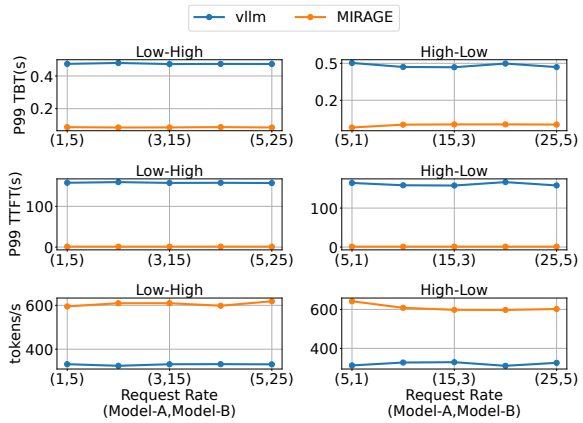


Figure 9: *Oneiros* outperforms vLLM on OPT-30b (Model-A) + OPT-6.7b (Model-B) with different request arrival rates.

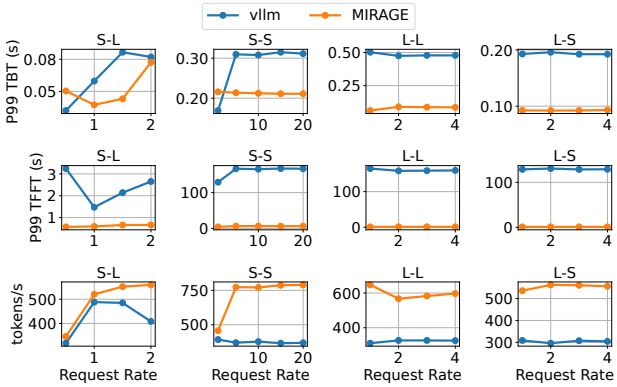


Figure 10: *Oneiros* outperforms vLLM on C2 across both short (S) and long (L) input requests.

processes to share internal GPU resources. It offers limited isolation since it does not partition components such as caches or memory controllers. (2) *Strict physical isolation*, exemplified by Multi-Instance GPU (MIG) [43], partitions the GPU into independent instances, providing strong physical isolation across workloads.

Non-Strict Physical Isolation: As shown in Figure 12, *Oneiros* effectively reduces both tail TBT and TTFT latency as request arrival rates increase toward peak load. On average, it achieves a 65.5%

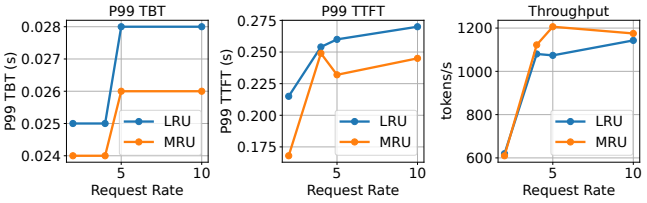


Figure 11: MRU (used in *Oneiros*) remaps models expected to be used furthest in the future, reducing tail latency and improving throughput compared to LRU.

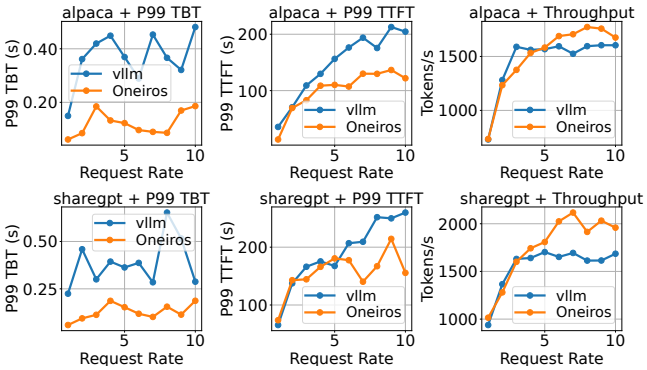
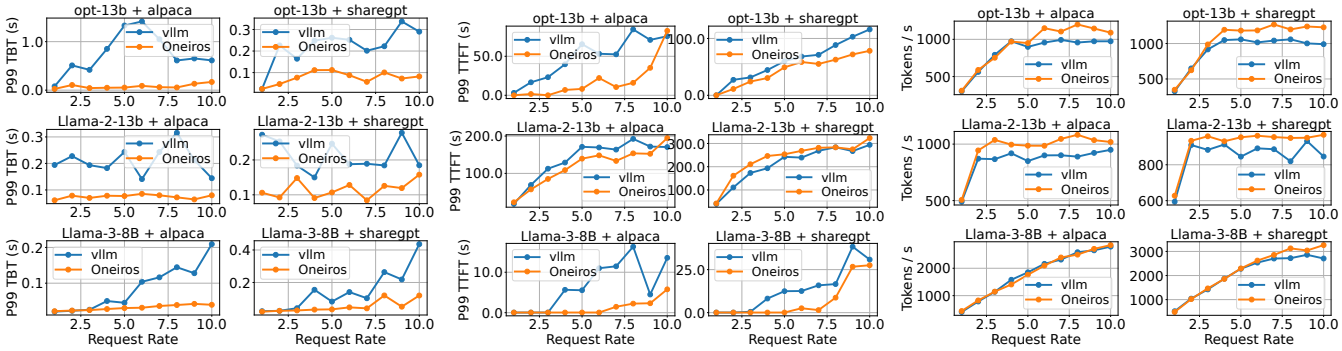


Figure 12: *Oneiros* achieves higher throughput and lower tail TBT latency than vLLM under non-strict physical isolation.

reduction in P99 TBT, a 20.7% reduction in P99 TTFT, and a 6.6% increase in throughput across varying arrival rates. These results highlight the robustness of *Oneiros*: it not only delivers strong performance under temporal GPU resource sharing but also offers significant improvements in spatial sharing configurations.

Strict Physical Isolation: *Oneiros* also maintains strong performance under strict physical isolation. As shown in Figure 13, it significantly reduces both tail TBT and TTFT latency as arrival rates increase, achieving an average 57.4% reduction in P99 TBT, 34.8% reduction in P99 TTFT, and a 7.9% increase in throughput across different models and load conditions. Notably, strict physical isolation between models can be viewed as a special case of single-model LLM inference. This further demonstrates the flexibility and robustness of *Oneiros* across diverse deployment scenarios.



(a) Tail TBT Latency. **(b) Tail TTFT Latency.** **(c) Throughput.**

Figure 13: Comparison between vllm and *Oneiros* under non-strict physical isolation. *Oneiros* achieves slightly higher throughput, but reduces tail TBT and TTFT latency significantly.

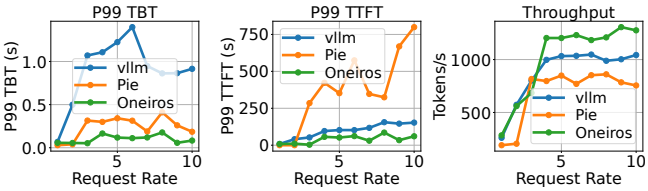


Figure 14: *Oneiros* reduces tail TBT and TTFT latency while delivering higher throughput than Pie.

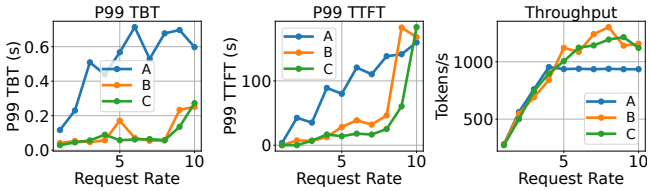


Figure 15: Using $m = \alpha + 2$ (B/C) yields lower latency and higher throughput than consistently using $m = \alpha + 1$ (A).

7.4 *Oneiros* vs KV Cache Swapping

Compared to KV-cache swapping, *Oneiros* achieves higher CPU-GPU bandwidth via unidirectional data transfer (§ 3.2), improving throughput. We evaluate *Oneiros* against Pie [73], a KV cache swapping based method. Figure 14 shows the tail latency and throughput for serving OPT-13b on the Alpaca dataset of vLLM, Pie, and *Oneiros*. GPU memory usage follows Table 1, consistent with § 7.5 and § 7.6. Compared to Pie, *Oneiros* reduces tail TBT and TTFT[6] latency by 35.0% and 93.6%, respectively, while improving throughput by 47.1%. This improvement is attributed to the ability of *Oneiros* to avoid the overhead of backing up KV cache to CPU memory, thereby enabling more efficient utilization of CPU-GPU bandwidth.

Beyond performance gains, *Oneiros* offers greater flexibility in temporal sharing scenarios than KV-cache–swapping approaches. In these scenarios, KV cache may be maintained only for models actively generating output tokens, while inactive models may not retain any. As a result, KV cache swapping based methodologies cannot reclaim memory from inactive models, limiting the ability to dynamically expand GPU memory for the KV cache of active models. *Oneiros* overcomes this limitation by repurposing the memory allocated to inactive models, even in the absence of KV cache.

7.5 Effectiveness of Our Layer Selection Strategy

The layer selection strategy is important for enabling *Oneiros* to effectively pipeline GPU computation with CPU-GPU parameter transfer, thereby minimizing performance overhead. The main trade-off involves selecting the number of layers (m, as defined in § 5.4) to transfer from CPU to GPU when α layers are remapped. Using $m = \alpha+1$ reduces the amount of data transfer, while $m = \alpha+2$ better addresses data dependencies by allowing full pipelining through a double-buffering mechanism. We present an empirical evaluation of three configurations: (A) using $m = \alpha + 1$ consistently, (B) using $m = \alpha + 2$ consistently, and (C) a dynamic scheme that selects $m = \alpha + 1$ for small α and switches to $m = \alpha + 2$ as α increases. Figure 15 presents the tail latency and throughput of serving the OPT-13b model on the Alpaca dataset using the different layer selection strategies described above. Our findings show that using $m = \alpha + 2$ – where transferred layers share a two-layer memory region to enable double buffering – is necessary for optimal performance. Using $m = \alpha + 1$ consistently (A) can result in a reduction in 12.7% throughput compared to using $m = \alpha + 2$ consistently (B) or switching to $m = \alpha + 2$ as α increases (C). The reduction in the amount of data transfer when using $m = \alpha + 1$ does not outweigh the overhead caused by disruptions to the GPU execution pipeline.

7.6 When and How Many Layers to Remap?

We conduct ablation studies to explore additional design parameter choices in *Oneiros*. While parameter remapping needs to be initiated only when KV cache capacity is exhausted, determining whether to "halt" remapping during off-peak periods is also important (§ 7.6.1). Moreover, more aggressive remapping can help avoid recomputation and reduce tail latency, but it may increase average latency due to increased data transfer overhead (§ 7.6.2).

7.6.1 When to halt remap? *Oneiros* monitors KV cache utilization at runtime and detects when free KV cache is available, typically during non-peak periods. For example, real-world traces show that request volume during non-peak times drops to approximately 20%

[6]We include the queuing delay in TTFT (TTFT = time from request arrival to generation of the first token = queuing delay + prefill time). Hence, by improving the throughput, *Oneiros* effectively reduces the queuing delay significantly, thereby reducing TTFT.

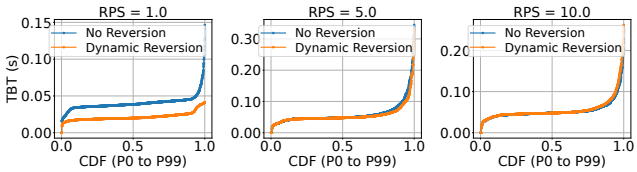


Figure 16: TBT CDF comparison between without and with *Dynamic Reversion* for OPT-13b+Alpaca. *Dynamic Reversion* effectively reduces TBT latency for non-peak periods.

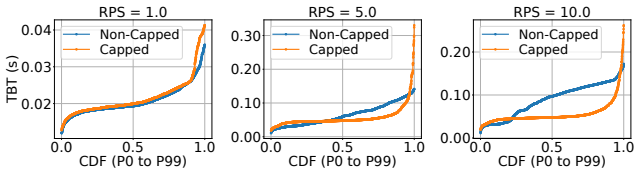


Figure 17: Capped vs non-capped *remapping percentage*. To prevent parameter loading from becoming the bottleneck, the *remapping percentage* must be appropriately capped.

of that seen during peak periods [60]. To optimize performance in such periods, *Oneiros* supports "halting" parameter remapping by reconfiguring the previously remapped memory space back for parameter usage. We name this mechanism *Dynamic Reversion* and evaluate it in this section.

Figure 16 compares the CDF of TBT latency for OPT-13B on the Alpaca dataset, with and without *Dynamic Reversion*. At peak load (10.0 RPS), *Dynamic Reversion* offers limited benefit since remapping is consistently required to avoid recomputation. During non-peak periods (1.0 RPS), however, it improves performance substantially—reducing P50 latency by 49% and consistently lowering TBT.

7.6.2 How many layers to remap? A key design parameter in *Oneiros* is the *remapping percentage* – the proportion of model parameter memory dynamically repurposed for KV cache. Higher percentages help sustain peak request loads that are hard to predict offline, but excessive remapping increases CPU–GPU transfer overhead, degrading concurrent request performance. We evaluate the impact of remapping percentage using OPT-13b with Alpaca dataset. As shown in Figure 17, aggressive remapping reduces tail latency under light workloads (e.g., 1.0 RPS), but increases average latency due to higher CPU-GPU data transfer overhead. Capping the remapping percentage yields a better balance—maintaining low average latency across workloads while still reducing tail latency. At 10 RPS, the capped strategy reduces P99 latency by 58% and P50 by 44%.

8 Discussion and Related Work

Compatibility with other LLM Optimization Solutions: *Oneiros* integrates seamlessly with state-of-the-art LLM inference optimizations. When the *Dynamic Remapping Engine* reaches its remapping percentage limit, it can leverage CPU offloading methods such as NEO [22] and FlexInfer [38] to maintain service quality. Unlike KV-cache swapping, *Oneiros* requires no CUDA kernel modifications, ensuring compatibility with advanced kernels like Pod-Attention [25] and LServe [74]. As a scheduler-independent memory engine, *Oneiros* supports temporal [4, 13, 14, 47, 71], spatial [6, 9, 16, 56, 61, 70, 72], and hybrid [10, 77] sharing policies. By activating parameter remapping only when GPU memory is insufficient for KV cache, *Oneiros* serves as a fallback for schedulers that mispredict memory demand under dynamic workloads.

Evolving LLM Trends: The growing popularity of multi-agent workflows [5, 17, 26, 31, 63] makes *Oneiros* well-suited for scenarios involving temporal GPU resource sharing. Furthermore, as the trend toward deploying multiple smaller-scale LLMs on a single host becomes more prevalent [68], *Oneiros* offers additional benefits by enabling efficient memory reuse across co-located models.

Evolving Hardware Trends: The Grace Hopper Superchip highlights a clear trend toward higher CPU-GPU bandwidth in future hardware. Its successor, GB200 [40], preserves the 900 GB/s bidirectional bandwidth while scaling to 2 GPUs per node and up to 72 GPUs in an NVLink-connected rack-scale system. This architecture ensures that *Oneiros* remains compatible with multi-GPU and distributed LLM inference optimizations [13, 62, 77]. The improvements in CPU DRAM bandwidth, further reinforces this trend. Unlike traditional DDR memory, the LPDDRX memory used in GH200 offers significantly higher bandwidth with lower power consumption, enabling such high CPU-GPU transfer rates [29, 32, 45].

Generic Applicability of *Oneiros* across GPU Types: *Oneiros* can also be deployed on earlier hardware generations, such as PCIe-based GPU systems with lower CPU-GPU bandwidth. The *Remapping Controller* in *Oneiros* dynamically determines the number of layers to remap based on the measured CPU-GPU transfer time per layer. On lower-bandwidth systems, *Oneiros* remains effective by adaptively remapping fewer layers (§ 5.3). In such environments, it can be further improved by leveraging CPU offloading (as discussed in § 3.1) to achieve additional performance gains.

Other Related Work: Although prior work has explored using CPU memory as an extension of GPU memory, the focus has predominantly been on training scenarios, such as DeepSpeed [52, 53, 55, 75] and others [21, 35, 69, 78]. Inference serving is latency-sensitive rather than throughput-oriented, making memory offloading particularly challenging. FlexGen [58] performs on-demand CPU–GPU data migration, often delaying computation as transfers complete [73]. Pie [73] was the first to exploit higher CPU–GPU bandwidth via KV-cache swapping, but its performance suffers from bidirectional transfer overhead. *Oneiros* differs fundamentally—serving as the first system to enable efficient *multi-tenant* LLM inference with dynamic memory remapping.

9 Conclusion

We presented *Oneiros*, a Dynamic Remapping Engine that elastically repurposes model parameter memory to expand KV-cache capacity for LLM inference, achieving low tail latency and high throughput. *Oneiros* exploits modern CPU–GPU bandwidth to overlap parameter transfers with computation, minimizing remapping overhead. It integrates seamlessly with existing serving systems and supports diverse GPU sharing and scheduling policies.

Acknowledgement: We thank the anonymous reviewers for their valuable feedback and Dimitrios Liakopoulos for proofreading. This research was supported in part by NSF grant numbers #2326894 and #2425655, the UT ECE junior faculty start-up fund, UT iMAGiNE consortium, an award from the UT Machine Learning Lab (MLL), the AMD Chair Endowment, the Amazon Research Award, and Vista GPU Cluster through CGAI and TACC at UT Austin.

References

[1] Muhammad Adnan, Akhil Arunkumar, Gaurav Jain, Prashant Nair, Ilya Soloveychik, and Purushotham Kamath. 2024. Keyformer: Kv cache reduction through key tokens selection for efficient generative inference. *Proceedings of Machine Learning and Systems* 6 (2024), 114–127.

[2] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming throughput-latency tradeoff in llm inference with sarathi-serve. In *OSDI*.

[3] Sohaib Ahmad, Hui Guan, Brian D Friedman, Thomas Williams, Ramesh K Sitaraman, and Thomas Woo. 2024. Proteus: A high-throughput inference-serving system with accuracy scaling. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 318–334.

[4] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. 2020. {PipeSwitch}: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 499–514.

[5] Chi-Min Chan, Weize Chen, Yusheng Su, Jianxuan Yu, Wei Xue, Shanghang Zhang, Jie Fu, and Zhiyuan Liu. 2023. Chateval: Towards better llm-based evaluators through multi-agent debate. *arXiv preprint arXiv:2308.07201* (2023).

[6] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. 2022. Serving heterogeneous machine learning models on {Multi-GPU} servers with {Spatio-Temporal} sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 199–216.

[7] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 153–167.

[8] Christina Delimitrou and Christos Kozyrakis. 2018. Amdahl's law for tail latency. *Commun. ACM* 61, 8 (2018), 65–72.

[9] Aditya Dhakal, Sameer G Kulkarni, and KK Ramakrishnan. 2020. Gslice: controlled spatial sharing of gpus for a scalable inference platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 492–506.

[10] Jiangfei Duan, Runyu Lu, Haojie Duanmu, Xiuhong Li, Xingcheng Zhang, Dahua Lin, Ion Stoica, and Hao Zhang. 2024. MuxServe: flexible spatial-temporal multiplexing for multiple LLM serving. *arXiv preprint arXiv:2404.02015* (2024).

[11] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[12] Pouya Esmaili-Dokht, Francesco Sgherzi, Valéria Soldera Girelli, Isaac Boixaderas, Mariana Carmin, Alireza Monemi, Adria Armejach, Estanislao Mercadal, Germán Llort, Petar Radojković, et al. 2024. A mess of memory system benchmarking, simulation and application profiling. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 136–152.

[13] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. Serverlessllm: Low-latency serverless inference for large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 135–153.

[14] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving {DNNs} like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 443–462.

[15] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, et al. 2020. Protean:{VM} allocation service at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 845–861.

[16] Bing-Shiun Han, Tathagata Paul, Zhenhua Liu, and Anshul Gandhi. 2024. KACE: Kernel-Aware Colocation for Efficient GPU Spatial Sharing. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*. 460–469.

[17] Shanshan Han, Qifan Zhang, Yuhang Yao, Weizhao Jin, Zhaozhuo Xu, and Chaoyang He. 2024. LLM multi-agent systems: Challenges and open problems. *arXiv preprint arXiv:2402.03578* (2024).

[18] Bagus Hanindhito, Bhavesh Patel, and Lizy K John. 2024. Bandwidth Characterization of DeepSpeed on Distributed Large Language Model Training. In *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 241–256.

[19] Md E Haque, Yong Hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S McKinley. 2015. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. *ACM Sigplan Notices* 50, 4 (2015), 161–175.

[20] Joel Hestness, Stephen W Keckler, and David A Wood. 2015. GPU computing pipeline inefficiencies and optimization opportunities in heterogeneous CPU-GPU processors. In *2015 IEEE International Symposium on Workload Characterization*. IEEE, 87–97.

[21] Chien-Chin Huang, Gu Jin, and Jinyang Li. 2020. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1341–1355.

[22] Xuanlin Jiang, Yang Zhou, Shiyi Cao, Ion Stoica, and Minlan Yu. 2025. Neo: Saving gpu memory crisis with cpu offloading for online llm inference. (2025).

[23] Shuowei Jin, Xueshen Liu, Qingzhao Zhang, and Z Morley Mao. 2024. Compute or load kv cache? why not both? *arXiv preprint arXiv:2410.03065* (2024).

[24] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for {μsecond-scale} Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 345–360.

[25] Aditya K Kamath, Ramya Prabhu, Jayashree Mohan, Simon Peter, Ramachandran Ramjee, and Ashish Panwar. 2024. Pod-attention: Unlocking full prefill-decode overlap for faster llm inference. *arXiv preprint arXiv:2410.18038* (2024).

[26] Jiin Kim, Byeongjun Shin, Jinha Chung, and Minsoo Rhu. 2025. The Cost of Dynamic Reasoning: Demystifying AI Agents and Test-Time Scaling from an AI Infrastructure Perspective. arXiv:2506.04301 [cs.LG] https://arxiv.org/abs/2506.04301 arXiv preprint.

[27] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 611–626.

[28] Munkyu Lee, Sihoon Seong, Minki Kang, Jihyuk Lee, Gap-Joo Na, In-Geol Chun, Dimitrios Nikolopoulos, and Cheol-Ho Hong. 2024. ParvaGPU: Efficient spatial GPU sharing for large-scale DNN inference in cloud environments. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[29] Sukhan Lee, Hyunyoon Cho, Young Hoon Son, Yuhwan Ro, Nam Sung Kim, and Jung Ho Ahn. 2018. Leveraging power-performance relationship of energy-efficient modern DRAM devices. *IEEE Access* 6 (2018), 31387–31398.

[30] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. {InfiniGen}: Efficient generative inference of large language models with dynamic {KV} cache management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 155–172.

[31] Junyou Li, Qin Zhang, Yangbin Yu, Qiang Fu, and Deheng Ye. 2024. More agents is all you need. *arXiv preprint arXiv:2402.05120* (2024).

[32] Shang Li, Dhiraj Reddy, and Bruce Jacob. 2018. A performance & power comparison of modern high-speed dram architectures. In *Proceedings of the International Symposium on Memory Systems*. 341–353.

[33] Xinyi Li, Sai Wang, Siqi Zeng, Yu Wu, and Yi Yang. 2024. A survey on LLM-based multi-agent systems: workflow, infrastructure, and challenges. *Vicinagearth* 1, 1 (2024), 9.

[34] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. 2023. {AlpaServe}: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 663–679.

[35] Gangmuk Lim, Jeongseob Ahn, Wencong Xiao, Youngjin Kwon, and Myeongjae Jeon. 2021. Zico: Efficient {GPU} memory sharing for concurrent {DNN} training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 161–175.

[36] Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, et al. 2024. Cachegen: Kv cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 38–56.

[37] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. 2024. Spotserve: Serving generative large language models on preemptible instances. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 1112–1127.

[38] Seonjin Na, Geonhwa Jeong, Byung Hoon Ahn, Aaron Jezghani, Jeffrey Young, Christopher J Hughes, Tushar Krishna, and Hyesoon Kim. 2025. FlexInfer: Flexible LLM Inference with CPU Computations. In *Proceedings of the 8th MLSys Conference*.

[39] Nvidia. 2025. Nvidia A100 GPU Architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[40] Nvidia. 2025. Nvidia GB200 GPU Architecture. https://www.nvidia.com/en-us/data-center/dgx-gb200/.

[41] Nvidia. 2025. Nvidia GH200 GPU Architecture. https://www.nvidia.com/en-us/data-center/grace-hopper-superchip/.

[42] Nvidia. 2025. Nvidia H100 GPU Architecture. https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper.

[43] Nvidia. 2025. NVIDIA Multi-Instance GPU. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/.

[44] Nvidia. 2025. NVIDIA Multi-Process Service. https://docs.nvidia.com/deploy/mps/index.html/.

[45] Sang-Soo Park, KyungSoo Kim, Jinin So, Jin Jung, Jonggeon Lee, Kyoungwan Woo, Nayeon Kim, Younghyun Lee, Hyungyo Kim, Yongsuk Kwon, et al. 2024. An lpddr-based cxl-pnm platform for tco-efficient inference of transformer-based large language models. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 970–982.

[46] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 118–132.

[47] Archit Patke, Dhemath Reddy, Saurabh Jha, Haoran Qiu, Christian Pinto, Chandra Narayanaswami, Zbigniew Kalbarczyk, and Ravishankar Iyer. 2024. Queue management for slo-oriented large language model serving. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*. 18–35.

[48] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. 2020. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 891–905.

[49] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2023. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems* 5 (2023).

[50] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. 2025. vAttention: Dynamic Memory Management for Serving LLMs without PagedAttention. In *ASPLOS*.

[51] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 325–341.

[52] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[53] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 1–14.

[54] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 3505–3506. doi:10.1145/3394486.3406703

[55] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. {Zero-offload}: Democratizing {billion-scale} model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 551–564.

[56] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. 2021. {INFaaS}: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 397–411.

[57] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX annual technical conference (USENIX ATC 20)*. 205–218.

[58] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*. PMLR, 31094–31116.

[59] Jovan Stojkovic, Pulkit A Misra, Íñigo Goiri, Sam Whitlock, Esha Choukse, Mayukh Das, Chetan Bansal, Jason Lee, Zoey Sun, Haoran Qiu, et al. 2024. SmartOClock: Workload-and risk-aware overclocking in the cloud. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 437–451.

[60] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. 2024. Dynamollm: Designing llm inference clusters for performance and energy efficiency. *arXiv preprint arXiv:2408.00741* (2024).

[61] Foteini Strati, Xianzhe Ma, and Ana Klimovic. 2024. Orion: Interference-aware, fine-grained GPU sharing for ML applications. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 1075–1092.

[62] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. 2024. Llumnix: Dynamic scheduling for large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 173–191.

[63] Yashar Talebirad and Amirhossein Nadiri. 2023. Multi-agent collaboration: Harnessing the power of intelligent llm agents. *arXiv preprint arXiv:2306.03314* (2023).

[64] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B Hashimoto. 2023. Stanford alpaca: An instruction-following llama model.

[65] ShareGPT Team. 2025. https://sharegpt.com/.

[66] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[67] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL]

[68] Fali Wang, Zhiwei Zhang, Xianren Zhang, Zongyu Wu, Tzuhao Mo, Qiuhao Lu, Wanjing Wang, Rui Li, Junjie Xu, Xianfeng Tang, et al. 2024. A comprehensive survey of small language models in the era of large language models: Techniques, enhancements, applications, collaboration with llms, and trustworthiness. *arXiv preprint arXiv:2411.03350* (2024).

[69] Qipeng Wang, Mengwei Xu, Chao Jin, Xinran Dong, Jinliang Yuan, Xin Jin, Gang Huang, Yunxin Liu, and Xuanzhe Liu. 2022. Melon: Breaking the memory wall for resource-efficient on-device machine learning. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*. 450–463.

[70] Tianyu Wang, Sheng Li, Bingyao Li, Yue Dai, Ao Li, Geng Yuan, Yufei Ding, Youtao Zhang, and Xulong Tang. 2024. Improving GPU Multi-Tenancy Through Dynamic Multi-Instance GPU Reconfiguration. *arXiv preprint arXiv:2407.13126* (2024).

[71] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. {AntMan}: Dynamic scaling on {GPU} clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 533–548.

[72] Fei Xu, Jianian Xu, Jiabin Chen, Li Chen, Ruitao Shang, Zhi Zhou, and Fangming Liu. 2022. igniter: Interference-aware gpu resource provisioning for predictable dnn inference in the cloud. *IEEE Transactions on Parallel and Distributed Systems* 34, 3 (2022), 812–827.

[73] Yi Xu, Ziming Mao, Xiangxi Mo, Shu Liu, and Ion Stoica. 2024. Pie: Pooling CPU Memory for LLM Inference. *arXiv preprint arXiv:2411.09317* (2024).

[74] Shang Yang, Junxian Guo, Haotian Tang, Qinghao Hu, Guangxuan Xiao, Jiaming Tang, Yujun Lin, Zhijian Liu, Yao Lu, and Song Han. 2025. LServe: Efficient Long-sequence LLM Serving with Unified Sparse Attention. *arXiv preprint arXiv:2502.14866* (2025).

[75] Jinghan Yao, Sam Ade Jacobs, Masahiro Tanaka, Olatunji Ruwase, Aamir Shafi, Hari Subramoni, and Dhabaleswar K Panda. 2024. Training ultra long context language model with fully pipelined distributed transformer. *arXiv preprint arXiv:2408.16978* (2024).

[76] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 521–538.

[77] Shan Yu, Jiarong Xing, Yifan Qiao, Mingyuan Ma, Yangmin Li, Yang Wang, Shuo Yang, Zhiqiang Xie, Shiyi Cao, Ke Bao, et al. 2025. Prism: Unleashing GPU Sharing for Cost-Efficient Multi-LLM Serving. *arXiv preprint arXiv:2505.04021* (2025).

[78] Tailing Yuan, Yuliang Liu, Xucheng Ye, Shenglong Zhang, Jianchao Tan, Bin Chen, Chengru Song, and Di Zhang. 2024. Accelerating the training of large language models using efficient activation rematerialization and optimal hybrid parallelism. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 545–561.

[79] Chen Zhang, Kuntai Du, Shu Liu, Woosuk Kwon, Xiangxi Mo, Yufeng Wang, Xiaoxuan Liu, Kaichao You, Zhuohan Li, Mingsheng Long, et al. 2025. Jenga: Effective Memory Management for Serving LLM with Heterogeneity. (2025), 446–461.

[80] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. 2022. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068* (2022).

[81] Kevin Zhao, Prateesh Goyal, Mohammad Alizadeh, and Thomas E Anderson. 2023. Scalable tail latency estimation for data center networks. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 685–702.