Elastic Memory 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, but its rapidly growing memory footprint makes GPU memory a primary bottleneck in modern serving systems. Recent approaches extend GPU memory using CPU memory through KV-cache swapping. However, because KV cache is continuously updated during decoding, swapping introduces substantial synchronization and bidirectional transfer overheads. We present *Oneiros*, a dynamic remapping engine for multi-tenant LLM serving. *Oneiros* is based on a simple observation: unlike KV cache, model parameters remain immutable during inference. Instead of swapping KV cache itself, *Oneiros* dynamically repurposes GPU memory allocated for model parameters as KV cache capacity, enabling non-blocking, unidirectional parameter transfer. This approach is particularly effective in multi-tenant environments, where memory allocated to inactive models can be reclaimed dynamically for active workloads. We implement *Oneiros* in vLLM and evaluate it on modern GH200 systems. Compared to vLLM, *Oneiros* reduces tail latency by up to 99.3% and improves throughput by up to 86.7%. Compared to KV-cache swapping approaches, *Oneiros* achieves substantially higher throughput by avoiding synchronization overheads during runtime memory extension. Source code of *Oneiros* is available at https://github.com/UT-SysML/Oneiros/[1].

1 Introduction

The memory demands of Large Language Models (LLMs) are growing much faster than GPU memory capacity, making memory a major bottleneck for efficient inference serving [3, 4, 8, 14–16, 26]. KV cache [8, 19] reduces inference latency by storing intermediate results from previously generated tokens, avoiding redundant computation during autoregressive decoding. As a result, modern LLM serving systems rely heavily on KV cache to sustain low latency and high throughput.

However, KV cache grows dynamically with sequence length and batch size. Once GPU memory is exhausted, existing systems must either recompute key-value pairs or extend GPU memory using CPU memory, both of which increase serving latency [8]. Figure 1a shows this effect in practice: when available KV cache becomes limited under higher request rates, recomputation causes tail latency to increase sharply. The problem becomes even more challenging in multi-tenant environments, where multiple LLMs compete for the same GPU memory resources [28].

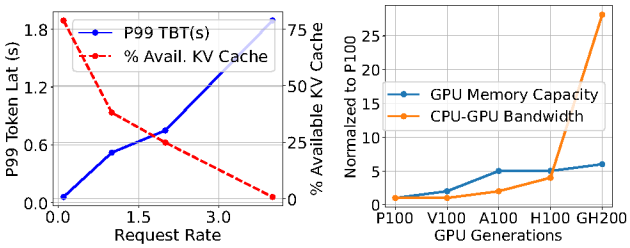
A natural way to alleviate GPU memory pressure is to use CPU memory as an extension of GPU memory [5, 6, 11, 13, 18, 22–24, 27]. Recent hardware trends make this increasingly practical. The NVIDIA Grace Hopper superchip (GH200) [15], for example, introduces a high-bandwidth NVLink-C2C interface between CPU and GPU, providing 7× the CPU-GPU bandwidth of H100 [16] and 14× that of A100 [14] (Figure 1b). This shift creates new opportunities for low-latency memory extension in LLM serving systems.

Existing approaches explore this opportunity in two main ways. Some systems offload portions of attention computation to CPUs, allowing KV cache to be generated and stored in CPU memory instead of GPU memory [6]. While this can support larger batch sizes, CPU computation itself can become the bottleneck, especially on newer systems where CPU-GPU bandwidth is no longer the dominant limitation. Other systems, such as Pie [27], swap KV cache between GPU and CPU memory, moving KV cache for future layers out of GPU memory and bringing it back when needed. This increases the effective KV cache capacity and reduces recomputation.

However, KV cache swapping introduces a different challenge. KV cache is continuously updated during decoding. As a result, swapping requires frequent synchronization between CPU and GPU memory. Figure 2 illustrates this dependency: KV cache in GPU memory cannot be reclaimed until the corresponding data has been written back to CPU memory. These bidirectional transfers introduce synchronization overheads

[1]This article summarizes the key ideas and findings from our full SoCC'25 paper on Oneiros [10].

[*]Equal contribution.



(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, creating new opportunities for memory extension.

that increase serving latency and reduce throughput.

In this paper, we explore a different approach. We make a simple observation: unlike KV cache, model parameters remain unchanged during inference. This difference fundamentally changes how memory extension can be performed. Instead of swapping KV cache itself, we propose *dynamic parameter remapping*[2], which repurposes a portion of GPU memory allocated for model parameters to support larger KV cache capacity at runtime.

Unlike KV cache swapping, parameter remapping is a non-blocking, unidirectional transfer. It does not require parameter writeback to CPU memory because model parameters remain immutable during inference. This substantially simplifies synchronization and memory management. Moreover, parameter remapping is particularly well suited for multi-tenant LLM serving. Models often remain idle for long periods without incoming requests [28]. In this work, we exploit these idle periods by reclaiming memory allocated to inactive models and repurposing it as KV cache for active workloads.

Realizing parameter remapping efficiently, however, requires solving several runtime systems challenges. The system must determine when remapping should begin or stop, which models should be targeted, how many layers should be remapped, and which layers should share GPU memory regions. These decisions depend on runtime load, request arrival dynamics, and input sequence lengths [17, 25, 28]. Poor remapping decisions can disrupt transfer-compute overlap, increasing latency and reducing throughput.

We present *Oneiros*, a *Dynamic Remapping Engine* for multi-tenant LLM inference serving systems. *Oneiros* dynamically and elastically adjusts KV cache capacity at layer granularity based on runtime demand. To support efficient remapping, *Oneiros* introduces a layer selection strategy that exploits the circular execution structure of autoregressive decoding to overlap parameter transfer with GPU computation

[2]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.

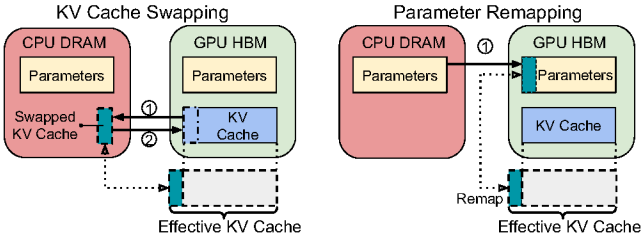


Figure 2: Both KV cache swapping and parameter remapping expand effective KV cache capacity by using CPU memory as an extension of GPU memory. KV cache swapping requires synchronization because KV cache is continuously updated during decoding. Parameter remapping instead uses unidirectional transfers with negligible synchronization overhead.

while minimizing CPU-GPU traffic. *Oneiros* integrates with existing LLM serving systems and remains compatible with different scheduling policies and GPU sharing mechanisms. We integrate *Oneiros* into vLLM and evaluate it on GH200 systems. Compared to vLLM, *Oneiros* improves throughput by 7.9%-86.7% and reduces tail latency by 20.7%-99.3%.

2 Background

This section provides background on modern LLM inference serving and the memory bottlenecks that motivate dynamic parameter remapping.

LLM Inference Serving. LLM inference serving consists of two phases: (1) the *prefill* phase, which processes the input sequence in a single pass, and (2) the *decode* phase, which autoregressively generates one output token at a time. Figure 3 illustrates the execution flow of a typical GPU-based serving system [8]. GPU memory is primarily consumed by two components: model parameters and KV cache [2, 8, 19, 20]. Before inference begins, model parameters are transferred from CPU memory to GPU memory. During decoding, KV cache stores intermediate results from previously generated tokens, avoiding redundant computation across token generations. As decoding progresses, KV cache grows dynamically with both sequence length and batch size. Modern serving systems therefore rely heavily on efficient KV cache management. Performance is typically characterized by TTFT (time-to-first-token), TBT (time-between-tokens), and throughput. TTFT captures responsiveness, TBT reflects token generation smoothness, and throughput measures the number of requests or tokens processed over time.

Memory Bottlenecks in LLM Serving. As model sizes continue to grow, GPU memory increasingly becomes the primary bottleneck in LLM serving. Memory pressure arises not only from model parameters, but also from KV cache, whose size depends on runtime factors such as request arrival patterns, batch size, and sequence length [2, 8].

Recent systems therefore focus heavily on KV cache management [1, 7–9, 12, 19, 20]. For example, vLLM uses *PagedAttention* [8] to dynamically manage KV cache pages during

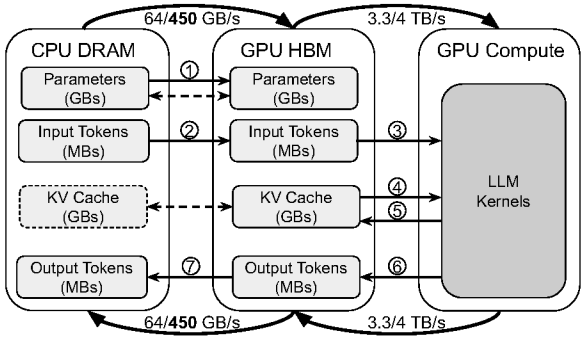


Figure 3: LLM inference serving on GPUs. Modern systems such as GH200 significantly increase CPU-GPU bandwidth, enabling CPU memory to act as a practical extension of GPU memory during inference.

runtime. However, even with these optimizations, KV cache usage remains difficult to predict ahead of time. When KV cache demand exceeds available GPU memory, systems must recompute key-value pairs, increasing serving latency and significantly affecting tail performance.

This challenge becomes more severe in multi-tenant environments, where multiple models contend for shared GPU memory resources and runtime KV cache demand fluctuates dynamically. As a result, preventing recomputation under memory pressure remains a central challenge in LLM inference serving systems.

3 Motivation

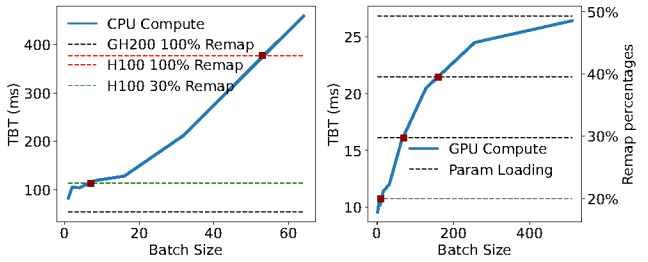
This section examines how modern high-bandwidth CPU-GPU systems change the design space for KV cache management in LLM inference serving. We compare three approaches for expanding effective KV cache capacity:

- **CPU offloading** [6], which offloads portions of attention computation and KV cache storage to CPU memory.
- **KV cache swapping** [27], which swaps KV cache between GPU and CPU memory during runtime.
- **Parameter remapping** (our approach), which dynamically repurposes GPU memory allocated for model parameters as KV cache capacity.

Modern systems such as GH200 provide sufficiently high CPU-GPU bandwidth to make runtime memory extension practical. However, efficiently exploiting this bandwidth requires minimizing synchronization overhead and carefully overlapping transfer with computation. As we show below, these requirements fundamentally favor parameter remapping over both CPU offloading and KV cache swapping.

3.1 Limitations of CPU Offloading

Prior work showed that offloading portions of attention computation to CPUs can increase effective KV cache capacity and support larger batch sizes [6]. This approach was particularly attractive on earlier systems where CPU-GPU



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

Figure 4: (a) On modern high-bandwidth systems, parameter transfer can become cheaper than CPU attention computation. (b) GPU computation time varies dynamically with runtime load, requiring remapping decisions to adapt online.

bandwidth was limited. However, modern systems such as GH200 shift the bottleneck. Figure 4a compares CPU computation time against parameter transfer time across different batch sizes. On a GH200, CPU-GPU transfer becomes fast enough that CPU computation itself emerges as the dominant limitation. This changes the tradeoff fundamentally. Rather than moving computation to CPUs, it becomes more efficient to continue executing on GPUs while dynamically transferring a subset of parameters. As CPU-GPU bandwidth continues to increase, parameter remapping becomes increasingly attractive.

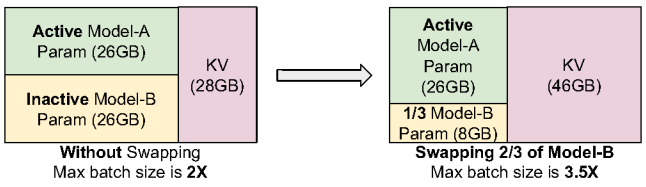
3.2 Limitations of Swapping KV Cache

KV cache swapping takes a different approach by moving KV cache itself between GPU and CPU memory [27]. Future-layer KV cache can be temporarily evicted from GPU memory and restored later when needed, increasing effective KV cache capacity and reducing recomputation. However, KV cache is continuously updated during decoding. As a result, swapping requires frequent synchronization between CPU and GPU memory. GPU memory cannot be reclaimed until KV cache contents have been safely written back to CPU memory. These bidirectional transfers disrupt transfer-compute overlap and reduce effective CPU-GPU bandwidth.

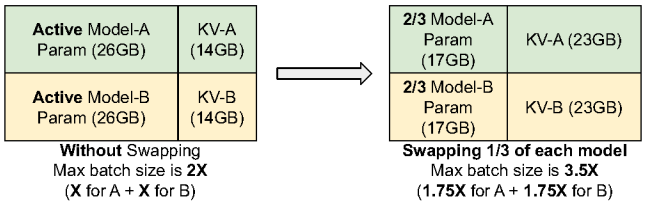
We observe this effect directly in bandwidth measurements. Unidirectional CPU-to-GPU transfers achieve substantially higher bandwidth than bidirectional transfer patterns. We measured that when reads and writes occur simultaneously, effective bandwidth drops by roughly 15%, reducing overall serving throughput. Thus, even though KV cache swapping exploits higher CPU-GPU bandwidth, synchronization overhead remains a fundamental limitation.

3.3 Our Proposal: Parameter Remapping

We instead propose *dynamic parameter remapping*, which repurposes GPU memory allocated for model parameters as KV cache capacity at runtime. Rather than swapping KV cache itself, the system dynamically remaps parameter memory to absorb periods of high KV cache demand. The key



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



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

Figure 5: Parameter remapping expands effective KV cache capacity in both temporal and spatial GPU sharing scenarios.

observation behind parameter remapping is that model parameters remain immutable during inference. Unlike KV cache, parameters do not require synchronization or writeback once transferred. As a result, parameter remapping uses non-blocking, unidirectional transfers that substantially simplify runtime memory management. Figure 5 illustrates this opportunity in both temporal and spatial GPU sharing scenarios. When two OPT-13b models share an H100 GPU, most GPU memory is occupied by model parameters, leaving limited space for KV cache. By dynamically reclaiming a portion of parameter memory, the system can significantly expand effective KV cache capacity and support larger runtime batch sizes.

Parameter remapping is particularly effective in multi-tenant serving environments. In production systems, many models remain idle for extended periods [28]. Our work exploits these idle intervals by reclaiming memory allocated to inactive models and repurposing it as KV cache for active workloads. In contrast, KV cache swapping cannot reclaim memory from inactive models when no KV cache is present.

3.4 Challenges in Parameter Remapping

Although parameter remapping avoids many of the synchronization costs of KV cache swapping, it introduces a new runtime systems problem: parameter transfer must be carefully coordinated with GPU execution. Ideally, parameter transfer overlaps completely with GPU computation. While one layer executes on the GPU, parameters for future layers are transferred concurrently from CPU memory. However, even with GH200-scale bandwidth, parameter transfer remains slower than fully on-demand execution. As a result, only a fraction of parameters can be remapped dynamically at runtime, while the remaining parameters must stay resident in GPU memory. We refer to this fraction as the *remapping percentage*. Efficient remapping therefore requires several runtime decisions:

When to remap? Remapping should be activated only when KV cache pressure becomes sufficiently high. Even fully pipelined transfer introduces overhead, making unnecessary remapping harmful during low-load periods.

Which models to remap? In multi-tenant systems, inactive models provide natural candidates for memory reclamation. However, overly aggressive remapping can delay future requests and increase tail latency.

How many layers to remap? The optimal remapping percentage depends on runtime load. Figure 4b shows that GPU computation time varies with batch size, while parameter transfer time remains relatively fixed. A static remapping policy therefore becomes inefficient across dynamic workloads.

Which layers to remap? The system must determine which layers remain resident in GPU memory and which are transferred dynamically. Poor layer selection disrupts transfer-compute overlap and reduces throughput. Efficient remapping therefore requires coordinated runtime memory management and layer scheduling.

4 *Oneiros* Design and Implementation

Oneiros is a dynamic remapping engine for multi-tenant LLM inference serving systems. It elastically expands KV cache capacity by repurposing GPU memory allocated to model parameters. Figure 6 details *Oneiros*'s workflow, which consists of three main components.

Metadata Store. The *Metadata Store* tracks model activity and memory utilization. It maintains information about active and inactive models, as well as KV cache usage and available GPU memory capacity.

Remapping Controller. The *Remapping Controller* determines when parameter remapping should occur and which models and layers should participate in remapping. When GPU memory pressure increases, the controller reclaims parameter memory to expand KV cache capacity. When KV cache demand decreases, the controller reverses the remapping and restores parameter memory. The controller operates dynamically at runtime and coordinates closely with the scheduler and memory allocator.

Async Transfer Engine. The *Async Transfer Engine* manages parameter transfer between CPU and GPU memory. When remapped parameters are needed for execution, the transfer engine asynchronously reloads them from CPU memory while GPU computation continues on other layers. Because autoregressive decoding executes layers in a deterministic order, *Oneiros* can overlap parameter transfer with GPU computation and minimize transfer overhead.

Figure 6 illustrates the runtime workflow of *Oneiros*. The scheduler selects active models for execution, while the memory allocator tracks parameter and KV cache usage. Using this runtime state, the *Remapping Controller* determines whether parameter remapping is necessary. If additional KV cache capacity is required, the controller updates memory mappings

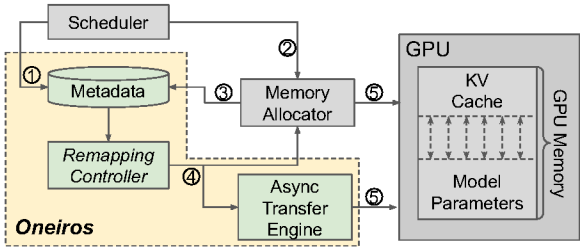


Figure 6: Overview of *Oneiros*.

and triggers asynchronous parameter transfer. The GPU then continues execution using the expanded KV cache space.

By dynamically reclaiming parameter memory at runtime, *Oneiros* enables existing serving systems to support larger batches and longer sequences while maintaining low synchronization overhead. We implement *Oneiros* in vLLM using asynchronous parameter transfer and virtual-memory-backed KV allocation based on vAttention [20].

5 Runtime Remapping

The key challenge in parameter remapping is that KV cache demand changes continuously at runtime. As request arrival rates, sequence lengths, and model activity fluctuate, the system must dynamically determine when remapping should begin, how aggressively it should proceed, and which layers should participate in remapping. These decisions directly affect transfer-compute overlap, synchronization overhead, and serving latency. *Oneiros* performs these decisions through the *Remapping Controller*. Figure 7 illustrates the runtime behavior of the controller across different phases of KV cache pressure. Depending on system load, the *Remapping Controller* may initiate remapping, reclaim additional parameter memory, or restore previously remapped memory back for parameter storage.

5.1 Runtime Remapping Policy

The *Remapping Controller* enables remapping only when KV cache demand exceeds available GPU memory. Although parameter remapping avoids the synchronization overheads of KV cache swapping, parameter transfer still introduces runtime overhead. As a result, remapping should occur only during periods of high KV cache pressure.

In multi-tenant environments, the *Remapping Controller* prioritizes reclaiming memory from inactive models before reclaiming memory from active ones. Under temporal GPU sharing, models scheduled for future execution are treated as inactive. Under spatial sharing, all models remain active and therefore participate jointly in remapping. When scheduler priorities are unavailable, *Oneiros* applies a Most Recently Used (MRU) policy to defer parameter reload costs as far into the future as possible.

The aggressiveness of remapping is determined dynamically at runtime. Efficient remapping requires parameter transfer to overlap with GPU computation; otherwise, transfer stalls become a new bottleneck. Let T_T denote the transfer time of a single layer and $T_{Compute}$ the available GPU computation window. To sustain overlap, the number of remapped layers N must satisfy:

$$T_T \cdot N \leq T_{Compute} \tag{1}$$

Because GPU computation time changes with runtime batch size and request load, the optimal remapping percentage also changes dynamically. Aggressive remapping may introduce unnecessary transfer overhead during low-load periods, while conservative remapping may fail to prevent recomputation during peak load. The *Remapping Controller* continuously adapts remapping decisions online to maintain efficient overlap between parameter transfer and GPU execution.

5.2 Circular Layer Scheduling

Selecting which layers to remap is equally important. The *Remapping Controller* must determine which layers remain resident in GPU memory and which layers share dynamically transferred memory regions. We guide this decision by exploiting the circular execution structure of autoregressive decoding. During token generation, layers execute repeatedly in the same order across successive decoding iterations. As a result, the final layer of one token generation is immediately followed by the first layer of the next.

Oneiros exploits this structure with a uniform-interval layer-selection strategy. Evenly spaced layers are selected to share GPU memory regions, maximizing the computation time between successive accesses to remapped layers and increasing opportunities to overlap parameter transfer with GPU execution. Figure 7 illustrates this behavior using Model-C with eight layers. When remapping a single layer, Layers 1 and 5 share a common GPU memory region while the remaining layers remain resident in GPU memory. In contrast, selecting Layers 1 and 8 would provide insufficient time to reload Layer 1 before it is needed again in the next decoding iteration.

We now formalize this intuition. Consider an n-layer model where α layers are remapped dynamically. Suppose m layers are transferred between CPU and GPU during each token generation, denoted by $L_1, L_2, \ldots, L_m$. Let T_T denote parameter transfer time for a single layer and T_c the GPU computation time for one layer. To fully overlap parameter transfer with computation:

$$T_T \leq \min(k_1 \times T_c, \cdots, k_{m-1} \times T_c, k_m \times T_c) \tag{2}$$

where k_i denotes the number of layers between consecutive remapped layers.

Maximizing the overlap window under the constraint $k_1 + k_2 + \cdots + k_m = n$ yields:

$$k_1 = k_2 = \cdots = k_m \tag{3}$$

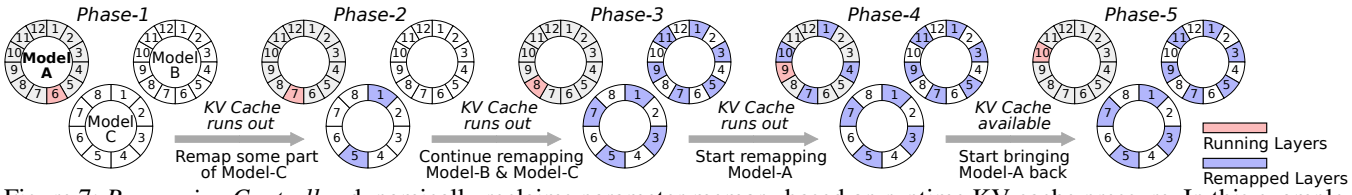


Figure 7: *Remapping Controller* dynamically reclaims parameter memory based on runtime KV cache pressure. In this example, Model-A is active while parameter memory from inactive models is progressively repurposed as KV cache capacity.

Thus, within the circular execution model, remapped layers should be evenly spaced throughout the network. This maximizes the minimum computation interval between successive remapped layers and therefore maximizes transfer-compute overlap.

The second design objective is to minimize CPU-GPU transfer traffic. When remapping pressure is low, remapped layers share a single GPU memory region to minimize transfer volume. As remapping pressure increases, *Oneiros* uses double buffering to better pipeline parameter transfer with execution:

$$T_T \times (\alpha + 2) \leq T_c \times n \tag{4}$$

This reduces pipeline stalls and improves throughput under heavier remapping loads.

Overall, the *Remapping Controller* continuously adapts remapping decisions at runtime while exploiting the circular structure of autoregressive execution to maximize transfer-compute overlap and minimize synchronization overhead.

6 Evaluation

We evaluate *Oneiros* on NVIDIA GH200 systems using Azure LLM inference traces, ShareGPT and Alpaca workloads, and multi-tenant model combinations built from OPT, Llama-2, and Llama-3 models. We compare against vLLM [8] and Pie [27], a recent KV-cache swapping system for GH200 platforms. Our evaluation focuses on tail latency (P99 TBT and TTFT) and throughput under runtime KV cache pressure.

6.1 *Oneiros* Reduces Tail Latency

Figure 8 evaluates *Oneiros* under multi-tenant GPU sharing environments using two representative model combinations. Across all workloads, *Oneiros* consistently reduces both TTFT and TBT tail latency while improving throughput relative to vLLM. Under high request rates, existing systems exhaust available KV cache and trigger recomputation, substantially increasing tail latency. By dynamically reclaiming parameter memory, *Oneiros* expands effective KV cache capacity and avoids these recomputation stalls. Across the evaluated workloads, *Oneiros* reduces tail TBT latency by 44.8%–82.5%, reduces tail TTFT latency by 74.8%–99.3%, and improves throughput by 39.9%–86.7%. These improvements persist across both temporal and spatial GPU sharing configurations, demonstrating that parameter remapping remains effective under diverse multi-tenant serving conditions.

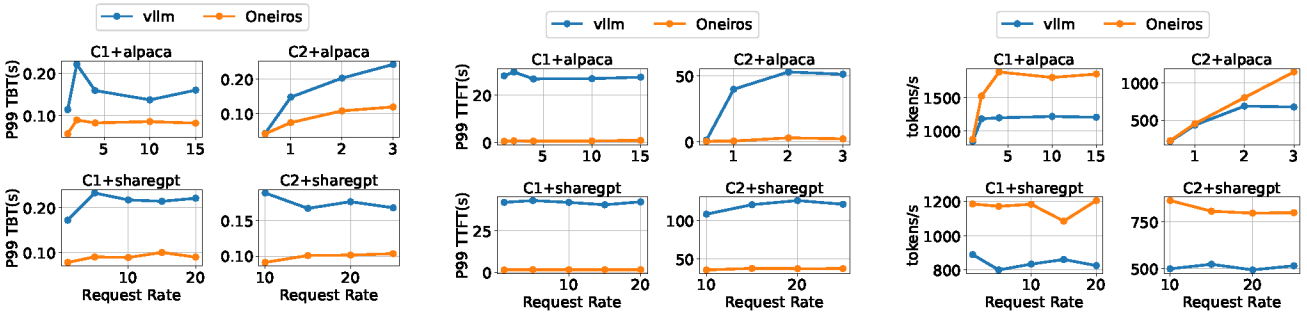
6.2 *Oneiros* vs. KV Cache Swapping

We next compare *Oneiros* against Pie [27], which expands KV cache capacity through KV-cache swapping. Figure 9 compares both systems on OPT-13b using the Alpaca dataset. Compared to Pie, *Oneiros* reduces tail TBT latency by 35.0%, reduces TTFT latency by 93.6%, and improves throughput by 47.1%. These improvements arise because parameter remapping avoids the synchronization and bidirectional transfer overheads of KV-cache swapping. Rather than continuously writing updated KV cache contents back to CPU memory, *Oneiros* relies on unidirectional parameter transfer, enabling more efficient overlap between transfer and GPU execution.

Parameter remapping also provides greater flexibility in multi-tenant serving environments. Inactive models may not retain KV cache that can be swapped out, limiting the effectiveness of KV-cache swapping approaches. In contrast, *Oneiros* can directly reclaim memory allocated to inactive models, even when no KV cache is present.

6.3 Runtime Overlap and Adaptivity

Figure 10 evaluates the layer scheduling strategy used by *Oneiros*. Using a single shared memory region minimizes transfer traffic but introduces additional pipeline stalls as remapping pressure increases. Double buffering better overlaps parameter transfer with GPU execution and improves throughput by up to 12.7%. We also evaluate the importance of dynamic remapping adaptation. During low-load periods, unnecessarily aggressive remapping can introduce transfer overheads that outweigh its benefits. *Oneiros* therefore dynamically restores remapped memory back for parameter storage when KV cache pressure subsides. This dynamic reversion mechanism reduces latency substantially during non-peak periods while preserving the ability to absorb KV cache pressure under peak load. Overall, the evaluation demonstrates that the primary benefits of *Oneiros* arise from three factors: avoiding recomputation under KV cache pressure, eliminating synchronization overheads associated with KV-cache swapping, and dynamically overlapping parameter transfer with autoregressive execution.



(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 to vLLM across multi-tenant workloads.

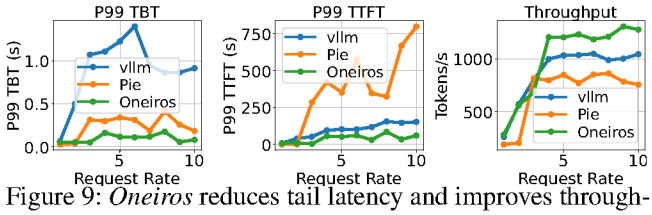


Figure 9: *Oneiros* reduces tail latency and improves throughput relative to Pie.

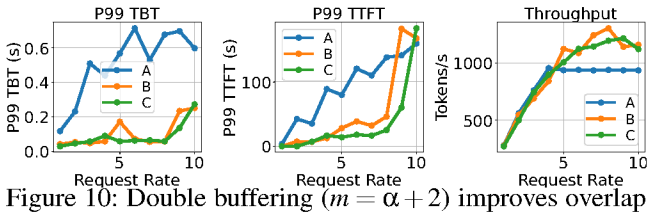


Figure 10: Double buffering ($m = \alpha + 2$) improves overlap between parameter transfer and GPU execution.

7 Discussion

Compatibility with Existing Serving Systems. *Oneiros* is designed as a scheduler-independent memory engine and integrates naturally with existing LLM serving systems. Because parameter remapping operates outside the attention kernel itself, it remains compatible with modern inference optimizations, including advanced attention kernels and existing GPU sharing policies. When remapping reaches its runtime limit, it can also complement CPU-offloading techniques by dynamically balancing transfer and computation overheads.

Evolving Hardware and Workload Trends. *Oneiros* reflects a broader shift in the assumptions underlying LLM serving systems. Emerging platforms such as GH200 and GB200 substantially increase CPU-GPU bandwidth, making runtime memory extension increasingly practical. At the same time, multi-tenant and multi-agent inference workloads continue to increase memory pressure through model co-location and dynamic request patterns. Parameter remapping exploits both trends by elastically reclaiming memory from inactive or underutilized models while avoiding the synchronization overheads of KV-cache swapping.

Applicability Across GPU Platforms. Although *Oneiros* is particularly effective on high-bandwidth systems, the approach remains applicable to lower-bandwidth GPU platforms as well. In such environments, the runtime controller adaptively reduces remapping aggressiveness to preserve transfer-compute overlap.

Relation to Prior Work. Prior work explored extending GPU memory through CPU offloading and KV-cache migration, primarily in training-oriented systems or single-model inference settings [21–24, 27]. *Oneiros* differs fundamentally by treating immutable model parameters as dynamically reclaimable memory capacity for multi-tenant inference serving. This shift eliminates many of the synchronization costs associated with KV-cache swapping while enabling memory reuse across co-located models.

8 Conclusion

We presented *Oneiros*, a dynamic remapping engine that elastically repurposes model parameter memory to expand KV-cache capacity for LLM inference serving. By exploiting the asymmetry between immutable model parameters and dynamically updated KV cache, *Oneiros* avoids many of the synchronization overheads associated with KV-cache swapping while overlapping parameter transfer with GPU execution. Our results show that parameter remapping substantially reduces tail latency and improves throughput under multi-tenant GPU sharing.

More broadly, *Oneiros* highlights how increasing CPU-GPU bandwidth fundamentally changes the design space of memory management for LLM serving systems. As future inference workloads become increasingly multi-tenant and memory-intensive, dynamic memory remapping provides a practical mechanism for improving memory elasticity without introducing substantial synchronization overhead.

References

[1] Muhammad Adnan, Akhil Arunkumar, Gaurav Jain, Prashant Nair, Ilya Soloveychik, and Purushotham Kamath. Keyformer: Kv cache reduction through key to-

kens selection for efficient generative inference. *Proceedings of Machine Learning and Systems*, 6:114–127, 2024.

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

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

[4] Bagus Hanindhito, Bhavesh Patel, and Lizy K John. Bandwidth characterization of deepspeed on distributed large language model training. In *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 241–256. IEEE, 2024.

[5] Chien-Chin Huang, Gu Jin, and Jinyang Li. 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*, pages 1341–1355, 2020.

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

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

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

[9] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. {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)*, pages 155–172, 2024.

[10] Ruihao Li, Shagnik Pal, Vineeth Narayan Pullu, Prasoon Sinha, Jeeho Ryoo, Lizy K John, and Neeraja J Yadwadkar. Oneiros: Kv cache optimization through parameter remapping for multi-tenant llm serving. In *Proceedings of the 2025 ACM Symposium on Cloud Computing*, pages 88–101, 2025.

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

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

[13] Seonjin Na, Geonhwa Jeong, Byung Hoon Ahn, Aaron Jezghani, Jeffrey Young, Christopher J Hughes, Tushar Krishna, and Hyesoon Kim. Flexinfer: Flexible llm inference with cpu computations. In *Proceedings of the 8th MLSys Conference*, 2025.

[14] Nvidia. Nvidia a100 gpu architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf, 2025.

[15] Nvidia. Nvidia gh200 gpu architecture. https://www.nvidia.com/en-us/data-center/grace-hopper-superchip/, 2025.

[16] Nvidia. Nvidia h100 gpu architecture. https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper, 2025.

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

[18] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. 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*, pages 891–905, 2020.

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

[20] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. vattention: Dynamic memory management for serving llms without pagedattention. In *ASPLOS*, 2025.

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

[22] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 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*, pages 1–14, 2021.

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

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

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

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

[27] Yi Xu, Ziming Mao, Xiangxi Mo, Shu Liu, and Ion Stoica. Pie: Pooling cpu memory for llm inference. *arXiv preprint arXiv:2411.09317*, 2024.

[28] Shan Yu, Jiarong Xing, Yifan Qiao, Mingyuan Ma, Yangmin Li, Yang Wang, Shuo Yang, Zhiqiang Xie, Shiyi Cao, Ke Bao, et al. Prism: Unleashing gpu sharing for cost-efficient multi-llm serving. *arXiv preprint arXiv:2505.04021*, 2025.