IMBALANCE-AWARE NUMA MEMORY PLACEMENT: A SIMPLE APPROACH TO MIXTURE-OF-EXPERTS SERVING ON TIGHTLY COUPLED ARCHITECTURES

Anonymous authors
Paper under double-blind review

ABSTRACT

Serving large Mixture-of-Experts (MoE) models is constrained by limited GPU memory capacity, which motivates offloading experts to CPU memory. Existing offloading systems are designed for PCIe-connected machines, where the interconnect is much slower than host memory. They migrate experts to the GPU or execute CPU-resident experts on the CPU. However, on tightly coupled architectures, the GPU can directly access CPU memory through NVLink-C2C at a bandwidth comparable to that of host memory. In this work, we propose a heterogeneous tensor that uses NUMA APIs to place both CPU-resident and GPU-resident experts within a single tensor. This allows GPU kernels to directly execute CPU-resident experts without explicit expert migration. We further propose **Imbalance-aware NUMA Memory Placement (INM)**, a simple static strategy that places each expert in CPU or GPU memory based on its precomputed access frequency. Since our approach only replaces the expert parameter tensor, we integrate it into vLLM and reuse its existing scheduler, KV cache management, and kernels. On GH200, our approach outperforms existing offloading systems in decode throughput for a single request at offload ratios of more than 50%. For batched requests, it achieves higher end-to-end throughput and lower latency than existing systems across all evaluated offload ratios. On the larger Qwen3.5-122B-A10B with 87.5% of experts offloaded, it achieves 1.27–1.46× higher decode throughput than previous research across diverse datasets. These results show that even a simple placement strategy enables efficient MoE serving on tightly coupled architectures.

1 INTRODUCTION

Sparse Mixture-of-Experts (MoE) models (Jacobs et al., 1991; Shazeer et al., 2017) have become a prominent architecture for scaling frontier generative AI models. MoE layers have a collection of E expert networks and a router that selectively assigns each input token to only a small subset of $k \ll E$ experts. Therefore, MoE models activate only a subset of their parameters during inference.

However, this computational sparsity does not eliminate the memory requirements. Without GPU memory management, all experts must reside in GPU memory. At the same time, the majority of the parameters in large-scale MoE models are concentrated in their MoE layers. For example, more than 97% of the parameters in DeepSeek-V4-Flash (DeepSeek-AI, 2026) are weights of the expert layers, and in most other MoE models, the MoE layers similarly account for more than 90% of the memory footprint. To address this mismatch between sparse computation and large model capacity, several previous systems have explored expert offloading, in which expert parameters are placed in host memory and transferred to or executed on GPUs as needed.

1.1 RELATED WORK

Several existing systems address the memory capacity problem of MoE models through expert offloading. EdgeMoE (Yi et al., 2025) and MoE-Infinity (Xue et al., 2025) keep the pool of experts in host memory and cache a subset of experts in GPU memory. These systems exploit expert activation

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

patterns to predict and prefetch the experts likely to be required and fetch cache-missed experts on demand.

Hybrid execution systems adopt a strategy of executing some experts on the CPU to reduce the amount of expert weights transferred to the GPU. Fiddler (Kamahori et al., 2025) executes some cache-missed experts on the CPU rather than transferring their weights to the GPU, thereby avoiding the corresponding GPU memory and data-movement overhead. ktransformers (Chen et al., 2025) puts hot experts in GPU memory and executes cold experts on the CPU. More recently, FreeToken (Yang et al., 2026) integrates expert caching, prefetching, and hybrid execution into an edge-native MoE serving system. During prefill, FreeToken employs layer-wise prefetching and overlaps expert transfers with computation. During decode, it dynamically partitions cache-missed experts between GPU execution and CPU execution according to the measured bandwidths of the CPU memory and the PCIe interconnect.

These system designs are fundamentally calibrated to hardware environments in which the interconnect is substantially slower than host memory. The measured one-way host-to-device (H2D) bandwidth for expert transfers over PCIe 5.0 $\times$16 is approximately 52 GB/s (Yang et al., 2026), which is often 7–9$\times$ lower than the bandwidth of host memory. Under such hardware constraints, minimizing data movement and executing part of the computation on the CPU becomes a central strategy for MoE serving.

1.2 Rethinking MoE Serving for Tightly Coupled Architectures

However, for cutting-edge heterogeneous machines, the fundamental assumption underlying these offloading approaches no longer necessarily holds. Tightly coupled architectures such as the NVIDIA Grace Hopper (GH200 (NVIDIA Corporation, 2023)) and Grace Blackwell (GB200 (NVIDIA Corporation, 2024)) connect the CPU and GPU through cache-coherent NVLink-C2C, providing substantially higher bandwidth than PCIe Gen5. These architectures employ a heterogeneous NUMA architecture in which CPU and GPU memory are exposed within a unified system address space, while remaining distinct memory regions. Moreover, the architecture based on the NVLink-C2C interconnect connecting the CPU and GPU will also be adopted in NVIDIA's next-generation Vera Rubin platform (NVIDIA Corporation, 2026), suggesting that these architectures are likely to become increasingly common in future accelerator systems. This architecture exhibits two fundamental characteristics that distinguish it from conventional PCIe-connected GPU systems.

Direct GPU access to CPU-resident data. The GPU on these architectures can directly access data residing in CPU memory through remote memory accesses because the CPU and GPU form a heterogeneous NUMA system with a unified address space. Consequently, GPU kernels can operate on CPU-resident data without explicitly copying it to GPU memory, effectively enabling zero-copy execution (Figure 1). The cache-coherent architecture introduces a new execution option of MoE: some experts can remain resident in CPU memory and be directly processed by the GPU.

The interconnect is as fast as host memory. NVLink-C2C can provide as high bandwidth as the host memory. Fusco et al. (2024) reported that measurements on the GH200 showed that the GPU can read remote LPDDR5X memory at 420.2 GB/s, while transfer benchmarks reported host-to-device and device-to-host bandwidths of 419.5 GB/s and 374.3 GB/s, respectively. For MoE offloading, this changes the nature of the bottleneck: the interconnect is no longer necessarily the slower component in the data path.

In this work, we show that zero-copy execution over NVLink-C2C enables a new direction for expert offloading, and that even a simple approach, which only decides where each expert resides, can achieve competitive performance on tightly coupled architectures. Our contributions are summarized as follows:

Zero-copy execution over NVLink-C2C. We apply the zero-copy execution capability of tightly coupled CPU-GPU architectures to LLM inference, allowing the GPU to directly execute experts residing in CPU memory without migrating them to GPU HBM. This introduces a different design point from conventional PCIe-based MoE serving systems that rely on expert migration and execution on CPU. The zero-copy execution can be realized by replacing the storage of existing `torch.Tensor` objects with memory regions allocated via `mmap` or `malloc`, making it straight-

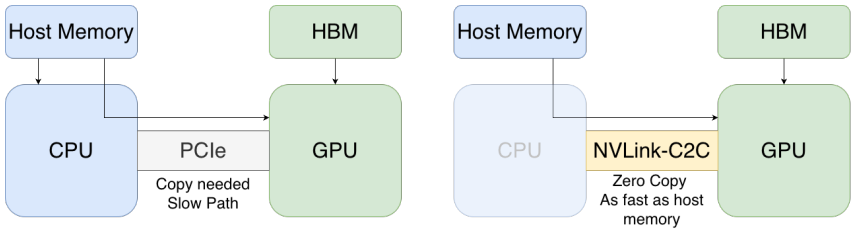


Figure 1: 1. Hybrid Execution copies experts through the slower PCIe path and executes some of them on the CPU. 2. Direct GPU Execution directly accesses and executes CPU-resident experts via NVLink-C2C.

forward to integrate with serving engines designed for batched requests, such as vLLM. We further evaluate existing approaches under concurrent batched requests and characterize how increased expert accesses across multiple requests affect the performance of existing approaches.

Imbalance-aware NUMA Memory Placement. We design and implement **Imbalance-aware NUMA Memory Placement (INM)**, a static expert placement strategy that places experts across GPU HBM and CPU memory according to their precomputed access frequencies. INM is intentionally simple: it only changes the physical memory placement of the expert parameters, without dynamic expert migration, prefetching, or custom kernels. Even with this simplicity, zero-copy execution over fast NVLink-C2C allows INM to achieve higher decode throughput and end-to-end performance. This demonstrates that tightly coupled architectures enable efficient MoE serving even with a minimal placement strategy.

2 METHOD

2.1 COUPLED ARCHITECTURE

Tightly coupled architectures integrate CPUs and GPUs via high-bandwidth interconnects. Although the CPU and GPU have physically separate memories in this architecture, they share a unified page table at the logical address level. Unlike conventional systems connected via PCIe, system memory allocated by `mmap` or `malloc` can be accessed by GPUs. In effect, both CPU and GPU memory can be used as memory accessible during GPU kernel execution. The CPU and GPU are exposed as NUMA nodes, allowing the physical placement of allocated memory to be controlled through NUMA APIs. This enables expert parameters to be explicitly placed in either CPU or GPU memory without changing their logical addresses.

Although NVLink-C2C provides higher bandwidth, we find that a single simple modification is still necessary to enable efficient MoE serving. Accesses to CPU memory from the GPU must traverse NVLink-C2C, whose bandwidth is comparable to host memory but still much lower than that of HBM. Therefore, carefully determining which data to place in CPU memory is still critical. Inspired by ktransformers, we determine the physical memory placement of MoE experts based on their imbalanced access frequencies. We implement **Imbalance-aware NUMA Memory Placement (INM)**, which exploits the non-uniform access patterns of MoE experts. We describe the implementation of INM in detail in the following section.

2.2 IMPLEMENTATION

2.2.1 EXPERT OFFLOADING IMPLEMENTATION

We implement expert offloading using the NUMA API. The key idea is to represent all experts as a single tensor while allowing individual experts to reside in either CPU or GPU memory. This allows the original MoE implementation to access experts without introducing additional custom implementations. We only change the physical data placement by NUMA APIs.

The experts are represented as a contiguous three-dimensional tensor of shape $(N_{\text{expert}}, d_{\text{hidden}}, d_{\text{intermediate}})$ in the implementations of vLLM (Kwon et al., 2023) and

Algorithm 1 Imbalance-aware NUMA Memory Placement (INM)

Require: Profiling dataset $\mathcal{D}$, offload ratio r
1: Count the router selections $c_{l,e}$ of each expert e in each layer l on $\mathcal{D}$
2: **for** each MoE layer l **do**
3: Select the $\lfloor rE \rfloor$ least frequently used experts as CPU-resident
4: Allocate the expert tensor with `mmap`
5: Bind each expert to the CPU or GPU NUMA node with `mbind`
6: Replace the original expert parameters with this tensor
7: **end for**
8: Serve requests with the unmodified serving engine

SGLang (Zheng et al., 2024). We allocate the contiguous memory using `mmap` and expose the allocated region as a PyTorch tensor using `torch::from_blob`. We independently control the residency of each expert by applying `mbind` to its corresponding memory range. `mbind` is a NUMA API function that determines which NUMA node physical memory is allocated on. We use `mbind` to specify whether each expert is placed in CPU or GPU memory by directly specifying the corresponding memory address. We refer to the resulting tensor as a *heterogeneous tensor* throughout the rest of this paper.

We replace the memory referenced by the original expert parameters with this heterogeneous tensor. This heterogeneous tensor causes an error in PyTorch's CUDA DeviceGuard. Therefore, for our experiments on GH200, we modify PyTorch to bypass this error and use custom builds of PyTorch 2.14.0 throughout the experiments. The implementation details for PyTorch, Triton, and vLLM are described in Appendix A.1. With these implementations, our approach can reuse the existing software components of vLLM, including its scheduler, KV cache management, and Triton kernels, by simply replacing the original expert parameter tensor with our heterogeneous tensor. Furthermore, we find that CUDA Graphs can be used with memory allocated by `mmap` on the GH200. Therefore, our implementation does not require additional static-buffer mechanisms for CUDA Graph compatibility, unlike existing offloading approaches (Yang et al., 2026). This allows the proposed expert offloading mechanism to be integrated into existing inference frameworks such as vLLM with minimal modifications.

2.3 Imbalance-aware NUMA Memory Placement

In this section, we describe how we determine the physical memory placement of our heterogeneous tensor. The distribution of expert accesses reportedly follows a power-law distribution (Yi et al., 2025). Under this distribution, experts can be broadly divided into those that are globally used and those that are rarely or never used (Liu et al., 2026). Based on this characteristic, we adopt a static placement policy similar to that used in ktransformers (Chen et al., 2025), in which expert access frequency determines the physical memory placement of each expert. We refer to this approach as **Imbalance-aware NUMA Memory Placement (INM)**. We deliberately adopt this simple and well-established policy, as our goal is to examine how far such a minimal strategy can go when combined with zero-copy execution on tightly coupled architectures.

Algorithm 1 summarizes INM. It consists of two phases. In the offline profiling phase, we run inference on a profiling dataset and count how many tokens are routed to each expert in each MoE layer. In the placement phase, for each layer, the least frequently accessed experts are placed in CPU memory according to the offload ratio, and the remaining experts are placed in GPU memory. The placement is then realized by constructing the heterogeneous tensor described in section A.1.2 and replacing the storage of the original expert parameters. After this initialization, the serving engine runs without any modification: INM introduces no runtime placement updates, expert transfers, or routing predictions, and therefore adds no runtime overhead to inference.

In batched serving, where diverse requests are processed concurrently, we expect that keeping globally frequently accessed experts in GPU memory is more effective than dynamically changing their placement. In our experiments, we use expert access frequencies precomputed on the TruthfulQA dataset (Lin et al., 2022), and evaluate on different datasets. We show that this static placement

achieves promising results despite its simplicity. The effect of dynamically updating the placement based on the observed access distribution is left for future work.

3 EXPERIMENT

3.1 COMPARISON OF EXPERT-WISE OFFLOADING

This section provides a controlled comparison: all methods are implemented on the same serving engine (vLLM) and evaluated on the same hardware (GH200). Differences in throughput can therefore be attributed to the offloading strategy itself rather than to differences in hardware or engine implementation. We compare the following four approaches:

- **Expert-wise Static Imbalance-aware NUMA Memory Placement (INM).** This is our main proposed approach. Data placed in CPU memory can be accessed and executed by the GPU without explicitly transferring the data to GPU memory. Expert placement is determined statically based on expert access frequencies precomputed using the TruthfulQA dataset (Lin et al., 2022).
- **Expert-wise Static Random Expert Placement.** In this approach, experts are randomly selected for placement in CPU memory. This experiment is designed to demonstrate the importance of the expert placement strategy.
- **Unified Virtual Address Offloading (UVA).** This is an offloading mechanism already implemented in vLLM that performs offloading at the module level. It allocates CPU memory that can be directly accessed by the GPU using `cudaHostGetDevicePointer()`. The key difference between this approach and ours is the granularity of offloading: while our method controls the placement of individual experts, this approach offloads the entire contiguous expert weight tensor.
- **Prefetch.** This is another offloading mechanism implemented in vLLM. It prefetches the experts of the next layer into GPU memory while the current layer is being executed. We adjust `group_size` and `num_in_group` to control the offloading ratio, while keeping `prefetch_step` fixed at 1 throughout the experiments. FreeToken (Yang et al., 2026) also partly employs this approach during prefill.

Figure 2 illustrates the difference among these methods. We use **Qwen3-30B-A3B-Thinking** (Yang et al., 2025), which contains 128 experts and activates 8 experts per token, and evaluate the methods on the **Massive Multitask Language Understanding (MMLU)** (Hendrycks et al., 2021) dataset with batch sizes of 16, 64. Varying the batch size allows us to investigate the effects of increasing the number of tokens processed during prefill and decode, as well as the effects of increased expert activation. Each request generates 128 tokens because we fix the output length to prevent decode throughput from fluctuating as the batch shrinks when some requests finish early. All following experiments use this setting on samples randomly selected from each dataset.

The results are shown in Figure 3. The x-axis represents the overall offload ratio, defined as the percentage of experts offloaded among the 6,144 experts in the model (128 experts $\times$ 48 layers). In prefill, both prefetch and our INM run at nearly the same speed with 25% of the experts offloaded as they do without offloading. Our INM exploits the skew in expert access when deciding which experts to offload and additionally benefits from zero-copy execution. However, when comparing the prefill results at batch sizes 16 and 64 across different offloading ratios, the throughput of INM degrades more sharply than that of prefetching as the batch size increases. This is because the larger number of tokens activates more experts, increasing CPU memory accesses and consequently introducing additional latency even with INM. In addition, as the workload becomes more compute-bound, prefetching becomes increasingly advantageous. In Qwen3-30B-A3B-Thinking, the experts of a single layer occupy about 1.21 GB, and transferring them takes about 2.68 ms even at the theoretical NVLink-C2C one-way bandwidth of 450 GB/s. As the amount of prefill computation increases, this transfer can be completed within the computation time.

Our INM consistently outperforms both random expert placement and UVA offloading across all batch sizes and offloading ratios of 25%, 50%, 75% for both prefill and decode. This is because INM reduces accesses to CPU memory through its expert placement. These results demonstrate the

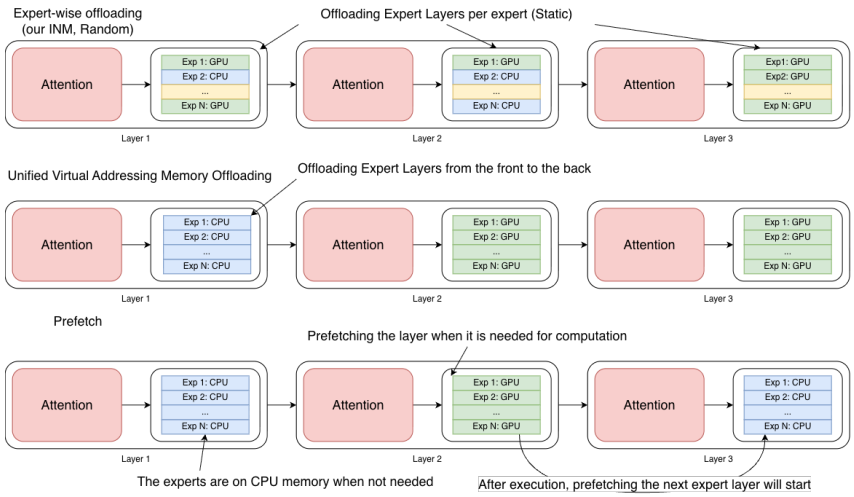


Figure 2: Illustration of the offloading methods evaluated in this work. The top shows expert-wise offloading used by our INM and the Random baseline, the middle shows UVA offloading, and the bottom shows prefetching.

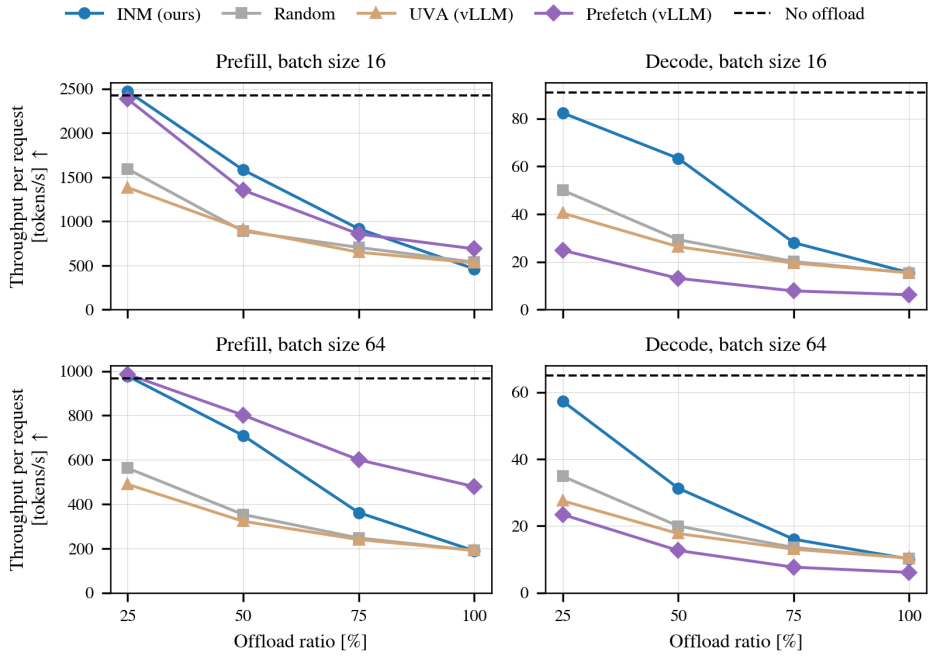


Figure 3: Prefill (left) and per-request decode (right) throughput of Qwen3-30B-A3B-Thinking on MMLU on GH200, with batch sizes of 16 (top) and 64 (bottom). The offload ratio is the fraction of all 6,144 experts placed in CPU memory; dashed lines indicate the no-offload baseline.

importance of an imbalance-aware placement strategy for effectively utilizing the available memory hierarchy.

Quantitatively, offloading 25% and 50% of the experts of Qwen3-30B-A3B-Thinking reduces GPU memory usage by 13.5 GiB and 27.0 GiB, respectively, and vLLM reuses this memory as KV cache, allocating an additional 147K and 295K tokens (1.54× and 2.07× the capacity without offloading). Since expert weights account for most of the parameters in MoE models, this benefit becomes larger for larger models, where the same offload ratio frees proportionally more GPU memory.

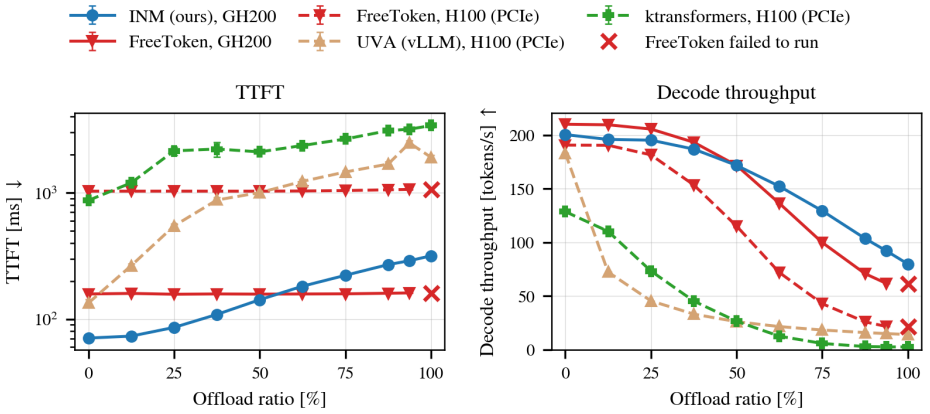


Figure 4: TTFT (left; log scale, lower is better) and decode throughput (right; higher is better) of Qwen3-30B-A3B-Thinking (BF16) at batch size 1 on Needle-in-a-Haystack (avg. 2,973 input tokens, 128 output tokens). Colors denote methods and line styles denote platforms (solid: GH200 with NVLink-C2C; dashed: x86-64 + H100 with PCIe Gen5). × indicates that FreeToken failed to run at 100% offload.

3.2 Comparison of Serving Engines

Next, we compare our method with the latest methods, FreeToken (Yang et al., 2026) and ktransformers (Chen et al., 2025). We evaluate the methods using the **Needle-in-a-Haystack** benchmark (Hengle et al., 2025) to ensure sufficiently long input sequences. We first compare TTFT and decode throughput with a batch size of one. ktransformers only works in an x86-64 + PCIe + H100 environment, so we measure it on a PCIe-based system. We use the `kt-expert-placement-strategy` option to specify the hot experts and determine which experts are placed in CPU memory, following the same placement strategy as ours. For FreeToken, we use measurement results from both the x86-64 + PCIe + H100 and GH200 environments under environments with different interconnect bandwidths and use the `moe-strategy` option with the `auto` setting. This setting automatically selects the strategy that the system determines to provide the best performance. To compare performance under the same offloading ratio, we specify the fraction of experts placed on the GPU using `kt-num-gpu-experts` for ktransformers and `moe-cache-rate` for FreeToken. Since FreeToken does not run with `moe-cache-rate` set to 0.0 (100% offloading), we mark this setting with × in the figures. Since vLLM's UVA offloading also relies on zero-copy execution, we additionally run it on the x86-64 + PCIe + H100 system to examine how zero-copy execution performs over PCIe. For our method, we use only the static expert offloading using our INM and do not perform prefetching during the prefill phase. Detailed hardware specifications are provided in the Appendix A.2.

Figure 4 shows that ktransformers and FreeToken (PCIe), both run on a PCIe-attached machine, perform worse than the two methods run on GH200 in both TTFT and decode throughput. In decode, FreeToken (PCIe) achieves about 190 tokens/s up to a 12.5% offload ratio, close to our INM and FreeToken (GH200) on GH200, but its performance degrades from 25% offload onward, possibly because, on PCIe machine, some experts begin to be executed on the CPU. ktransformers achieves lower decode throughput than FreeToken (PCIe) and the methods on GH200 even without offloading, suggesting that this gap stems from overheads in its integration with SGLang rather than from its offloading strategy. These results indicate that the gap in interconnect bandwidth is critical. Executing experts on the CPU mitigates the PCIe bottleneck only to a limited extent, and our simple static placement on GH200 outperforms these PCIe-based methods.

Next, we compare our INM and FreeToken (GH200), both run on GH200. In prefill, our INM achieves better TTFT up to a 50% offload ratio, and FreeToken (GH200) is better beyond 50%. Because FreeToken's prefill always prefetches the next layer regardless of the offload ratio, its TTFT stays constant, while our INM falls behind as the offload ratio increases and the computation time and data transfer between NVLink-C2C per layer grow. Nevertheless, at 93.75% offload, our INM

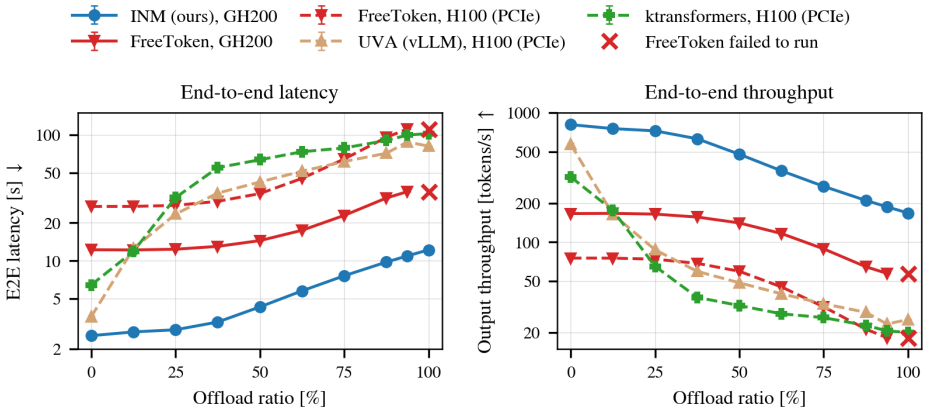


Figure 5: End-to-end latency (left; lower is better) and output throughput (right; higher is better) at batch size 16 on Needle-in-a-Haystack (47,562 input tokens per batch, 128 output tokens per request), both on log scales. E2E latency is the time to complete all 16 requests in a batch, and throughput is the total number of output tokens divided by this latency.

reaches a TTFT of 289 ms, about 1.8× that of FreeToken (GH200), yet still lower than those of the methods on the PCIe machine. In decode, our INM outperforms FreeToken for offload ratios of more than 50%. During decode, FreeToken manages the experts resident on the GPU with an LRU cache. As the offload ratio increases, the cache is updated more frequently, and the resulting transfer overhead is incurred at every decode step. The function that FreeToken uses to update its LRU cache is tuned for PCIe and achieves only about 201.8 GB/s for transfer, less than half of the 450 GB/s unidirectional bandwidth of NVLink-C2C. In contrast, our INM uses zero-copy execution, in which the efficient fused-MoE kernel reads expert weights directly from host memory, allowing it to serve Qwen3-30B-A3B-Thinking at 91.96 tokens/s even with 93.75% of the experts offloaded. At 100% offload, where every activated expert resides in host memory, each decode step reads all 384 activated experts (8 experts × 48 layers, 9.44 MB each), or about 3.62 GB, over NVLink-C2C. Our INM still achieves 79.79 tokens/s in this setting, that is, 12.53 ms per step. Even if the entire step were spent on these reads, the kernel would need to sustain at least 3.62 GB / 12.53 ms ≈ 289 GB/s, well above the 201.8 GB/s achieved by FreeToken's LRU-cache update path. Our method only changes where the expert weights reside, so it can directly reuse vLLM's Triton kernel, which is already optimized for memory loads, and thereby achieves high inference performance. In contrast, for vLLM UVA (PCIe), decode throughput falls below that of both ktransformers (PCIe) and FreeToken (PCIe) as soon as offloading begins. At 75% offload, vLLM UVA (PCIe) achieves 18.24 tokens/s, while FreeToken (PCIe) achieves 42.78 tokens/s. Each decode step must read about 2.72 GB of CPU-resident experts (75% of the 3.62 GB activated across all layers), which takes 42.4 ms over PCIe Gen5 at 64 GB/s and bounds throughput at 23.6 tokens/s. FreeToken (PCIe) exceeds this bound by executing cold experts on the CPU, while vLLM UVA (PCIe) remains limited by PCIe bandwidth. These results indicate that the high CPU–GPU bandwidth of NVLink-C2C is the key enabler of our approach. It allows even a simple static placement to outperform prior MoE offloading systems.

3.3 End-to-End Performance with Batch Requests

Next, we compare the performance under batch requests. We increase the batch size to 16 and compare the end-to-end (E2E) latency and throughput for 128 generated tokens. Although FreeToken supports multiple requests, it does not support streaming for batch requests. Therefore, TTFT and decode throughput cannot be measured for FreeToken in this setting, so we instead evaluate the E2E performance.

Figure 5 shows the E2E performance with a batch size of 16. Our approach achieves higher E2E throughput and lower E2E latency than other methods across all evaluated offload ratios. Our approach is implemented on vLLM and can therefore efficiently handle batched requests, achieving

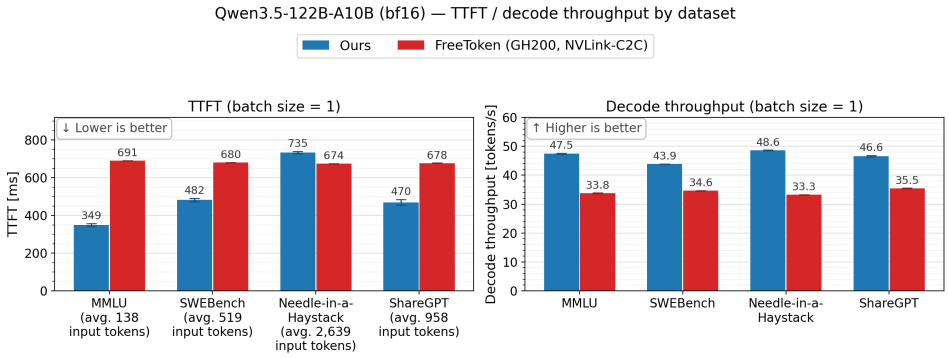


Figure 6: TTFT and decode throughput of Qwen3.5-122B-A10B on MMLU, SWE-bench, Needle-in-a-Haystack, and ShareGPT, covering different dataset distributions and model scalability.

better efficiency than existing approaches in such workloads. Although ktransformers is implemented on SGLang, which is well-suited for batched requests, it executes experts placed in CPU memory on the CPU. Therefore, as the batch size increases and the number of CPU-executed experts grows, its performance degrades substantially, as observed from the results with 12.5% offloading. Similarly, in FreeToken, batched requests increase the number of expert accesses, leading to more cache misses and consequently more transfers of cache-missed experts to the GPU. Notably, across all offload ratios from 0% to 93.75%, INM achieves 3.0–4.8× lower E2E latency than FreeToken (GH200). This consistent advantage stems from the integration of our approach into vLLM which allows it to directly inherit vLLM's efficient batched execution.

3.4 Scalability across Model Sizes and Datasets

To show that our method remains effective across different models and datasets, we evaluate the larger Qwen3.5-122B-A10B model (Qwen Team, 2026) with 87.5% of experts offloaded, measuring TTFT and decode throughput (tokens/s) for each dataset. We select the datasets to comprehensively evaluate how content diversity and input length affect our method. MMLU (Hendrycks et al., 2021) requires reasoning over text from a wide range of subjects. SWE-bench (Jimenez et al., 2024) measures software engineering skills, and we expect its narrow domain to skew expert usage toward specific experts. Needle-in-a-Haystack (Hengle et al., 2025) consists of long inputs. ShareGPT (Aeala, 2023) contains real-world conversations between users and a chatbot. The results are shown in Figure 6. For TTFT, our method's latency grows from 349ms at 138 tokens to 735ms at 2,640 tokens, while FreeToken remains nearly constant at approximately 675–690ms regardless of input length. Consistent with our earlier analysis, our method slightly underperforms FreeToken in TTFT on datasets with long inputs. However, for decoding, our method achieves approximately 1.27–1.46× higher throughput than FreeToken across all datasets, demonstrating that its effectiveness holds across different models and datasets.

4 Conclusion

We proposed a heterogeneous tensor and imbalance-aware NUMA memory placement (INM) for tightly coupled architectures. By applying them to MoE experts within this tensor, we made CPU memory directly usable by GPU kernels. We integrated this approach into vLLM, allowing us to reuse its existing effective software stack. With zero-copy execution and INM, our approach outperforms existing methods in decode throughput for a single request at offload ratios of more than 50%. For batched requests, we showed that existing MoE offloading approaches are not well suited, and our approach outperforms them. We also confirmed that our approach remains effective for a larger model and across diverse datasets. Our approach still has a limitation. Its prefill performance degrades at high offload ratios and with long inputs. We expect that techniques such as prefill-decode disaggregation can mitigate this limitation. We believe this work is a first step toward MoE offloading that exploits CPU memory on tightly coupled architectures.

AI USE STATEMENT

In this work, we used generative AI tools to assist with or polish writing, for information retrieval and discovery, including related work, and to draft sections of the paper. We have not used generative AI tools for research ideation or research execution, generating synthetic datasets, or proving mathematical claims. The remaining required disclosure tasks are not applicable to this work. For example, EdgeMoE (Yi et al., 2025) was identified with the assistance of generative AI as a relevant work related to MoE offloading. We reviewed the original paper and verified the relevant information before incorporating it into our related work. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We have taken several measures to facilitate the reproducibility of our work. The method section 2 describes the implementation and design of our approach, while the appendix A.1 provides further implementation details, including the versions of the software libraries used, code examples of the key implementation components, and explanations of the underlying implementation concepts. The experiment section 3 explicitly reports the models and datasets used in our evaluation, as well as the experimental settings. Together, these sections provide the implementation and experimental information necessary to reproduce our results.

REFERENCES

Aeala. Sharegpt_vicuna_unfiltered. https://huggingface.co/datasets/Aeala/ShareGPT_Vicuna_unfiltered, 2023.

Hongtao Chen, Weiyu Xie, Boxin Zhang, Jingqi Tang, Jiahao Wang, Jianwei Dong, Shaoyuan Chen, Ziwei Yuan, Chen Lin, Chengyu Qiu, Yuening Zhu, Qingliang Ou, Jiaqi Liao, Xianglin Chen, Zhiyuan Ai, Yongwei Wu, and Mingxing Zhang. Ktransformers: Unleashing the full potential of cpu/gpu hybrid inference for moe models. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, 2025.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026.

Luigi Fusco, Mikhail Khalilov, Marcin Chrapek, Giridhar Chukkapalli, Thomas Schulthess, and Torsten Hoefler. Understanding data movement in tightly coupled heterogeneous systems: A case study with the grace hopper superchip, 2024. URL https://arxiv.org/abs/2408.11556.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021.

Amey Hengle, Prasoon Bajpai, Soham Dan, and Tanmoy Chakraborty. Multilingual needle in a haystack: Investigating long-context behavior of multilingual large language models. In Luis Chiruzzo, Alan Ritter, and Lu Wang (eds.), *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 5165–5180, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics. ISBN 979-8-89176-189-6. doi: 10.18653/v1/2025.naacl-long.267. URL https://aclanthology.org/2025.naacl-long.267/.

Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87, 1991.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Keisuke Kamahori, Tian Tang, Yile Gu, Kan Zhu, and Baris Kasikci. Fiddler: Cpu-gpu orchestration for fast inference of mixture-of-experts models, 2025. URL https://arxiv.org/abs/2402.07033.

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*, pp. 611–626, 2023.

Stephanie Lin, Jacob Hilton, and Owain Evans. Truthfulqa: Measuring how models mimic human falsehoods, 2022. URL https://arxiv.org/abs/2109.07958.

Zijie Liu, Jie Peng, Jinhao Duan, Zirui Liu, Kaixiong Zhou, Mingfu Liang, Luke Simon, Xi Liu, Zhaozhuo Xu, and Tianlong Chen. A replicate-and-quantize strategy for plug-and-play load balancing of sparse mixture-of-experts llms, 2026. URL https://arxiv.org/abs/2602.19938.

NVIDIA Corporation. Nvidia gh200 grace hopper superchip platform. https://www.nvidia.com/en-us/data-center/grace-hopper-superchip/, 2023. Accessed: 2026-09-16.

NVIDIA Corporation. Nvidia gb200 grace blackwell superchip. https://www.nvidia.com/en-us/data-center/gb200-nvl72/, 2024. Accessed: 2026-09-16.

NVIDIA Corporation. Nvidia vera rubin platform. https://www.nvidia.com/en-us/data-center/technologies/rubin/, 2026. Accessed: 2026-09-16.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL https://qwen.ai/blog?id=qwen3.5.

Gabin Schieffer, Jacob Wahlgren, Jie Ren, Jennifer Faj, and Ivy Peng. Harnessing integrated cpu-gpu system memory for hpc: a first look into grace hopper. In *Proceedings of the 53rd International Conference on Parallel Processing*, pp. 199–209, 2024.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer, 2017. URL https://arxiv.org/abs/1701.06538.

Leyang Xue, Yao Fu, Zhan Lu, Luo Mai, and Mahesh Marina. Moe-infinity: Efficient moe inference on personal machines with sparsity-aware expert cache, 2025. URL https://arxiv.org/abs/2401.14361.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Shuo Yang, Xiaoze Fan, Melissa Pan, Haocheng Xi, Zhe Wang, Shanlin Sun, Kurt Keutzer, Song Han, Matei Zaharia, Chenfeng Xu, and Ion Stoica. Freetoken: Efficient edge-native moe serving with bandwidth-adaptive execution. *arXiv preprint arXiv:2608.16157*, 2026.

Rongjie Yi, Liwei Guo, Shiyun Wei, Ao Zhou, Shangguang Wang, and Mengwei Xu. Edgemoe: Empowering sparse large language models on mobile devices, 2025. URL https://arxiv.org/abs/2308.14352.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 62557–62583. Curran Associates, Inc., 2024. doi: 10.52202/079017-2000. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/724be4472168f31ba1c9ac630f15dec8-Paper-Conference.pdf.

A Appendix

A.1 Implementation Details

A.1.1 Making CPU Devices Executable on GPUs in PyTorch

This section briefly describes the modifications to PyTorch, Triton, and vLLM referenced in Section 2.

The versions of the libraries used in our experiments are PyTorch 2.14.0, vLLM 0.27.1, and Triton 3.6.0. We cloned the corresponding versions from their GitHub repositories and modified them as described below.

The GH200 supports executing CUDA kernels that access memory allocated by system-level mechanisms such as `mmap` and `malloc`. However, we found that PyTorch's `cudaPointerGetAttributes` in `pytorch/aten/src/ATen/cuda/CUDADevice.h` triggers an assertion and terminates the process when the pointer is not managed by CUDA. To address this issue, we maintain a hash set of pointers that are known to be safe to bypass this check and register them through a PyTorch C++ Extension. The relevant implementation is as follows. The current implementation supports only a single GPU and therefore implicitly assumes `device_index = 0`. This assumption will need to be modified to support multiple GPUs in future extensions.

```
namespace extension {

    TORCH_API void registerNUMAPointer(void*);
    TORCH_API void unregisterNUMAPointer(void*);
    TORCH_API bool isNUMAPointer(void*);

}

namespace at::cuda {

    inline Device getDeviceFromPtr(void* ptr) {
        cudaPointerAttributes attr{};

        AT_CUDA_CHECK(cudaPointerGetAttributes(&attr, ptr));
        if (extension::isNUMAPointer(ptr)){
            return {c10::DeviceType::CUDA, 0}; // device_index = 0
        }

        #if !defined(USE_ROCM)
        TORCH_CHECK(attr.type != cudaMemoryTypeUnregistered,
            "The specified pointer resides on host memory and
            is not registered with any CUDA device.");
        #endif

        return {c10::DeviceType::CUDA,
        static_cast<DeviceIndex>(attr.device)};
    }

} // namespace at::cuda
```

By using `TORCH_API`, these functions are exported from the PyTorch build and can be called from an external C++ extension, allowing the extension to access the same pointer registry. The implementations of `registerNUMAPointer(void*)`, `unregisterNUMAPointer(void*)`, and `isNUMAPointer(void*)` are shown below. Internally, we use a standard C++ `unordered_set` to maintain the registered pointers.

```
#include <ATen/cuda/CUDADevice.h>
#include <unordered_set>
```

```
namespace {

    std::unordered_set<void*> numa_pointers;

}

namespace extension {

    void registerNUMAPointer(void* ptr) {
        numa_pointers.insert(ptr);
    }

    void unregisterNUMAPointer(void* ptr) {
        numa_pointers.erase(ptr);
    }

    bool isNUMAPointer(void* ptr) {
        return numa_pointers.find(ptr) != numa_pointers.end();
    }

}
```

Triton also required a modification to handle pointers allocated by `mmap`. Specifically, we bypassed the device check in `triton/backends/nvidia/driver.py` in Triton 3.6.0. The specific modification is shown below.

```
uint64_t dev_ptr;
int status = cuPointerGetAttribute(&dev_ptr,
CU_POINTER_ATTRIBUTE_DEVICE_POINTER, ptr_info.dev_ptr);
goto cleanup; // Ignore all the following pointer checks.
if (status == CUDA_ERROR_INVALID_VALUE) {{
    PyErr_Format(PyExc_ValueError,
    "Pointer argument (at %d) cannot be
    accessed from Triton (cpu tensor?)", idx);
    ptr_info.valid = false;
} else if (status != CUDA_SUCCESS) {{
    CUDA_CHECK(status);  // Catch any other cuda API errors
    ptr_info.valid = false;
}
ptr_info.dev_ptr = dev_ptr;
cleanup:
Py_XDECREF(ret);
return ptr_info;
```

This completes the modifications to PyTorch and Triton. The implementation of the heterogeneous tensor is described in the next section.

A.1.2 Implementation of a heterogeneous tensor

We implement the heterogeneous expert tensor as a C++ extension for PyTorch. The implementation first allocates a contiguous virtual address range using `mmap`. We request transparent huge pages for this region using `madvise` with `MADV_HUGEPAGE`. The memory is then divided into contiguous regions corresponding to individual experts, and `mbind` is used to bind each region to either the CPU or GPU NUMA node according to its target residency.

The resulting memory region is wrapped as a PyTorch tensor using `torch::from_blob`, which allows the tensor to directly reference the allocated memory without introducing an additional copy. The following code shows the essential part of our implementation.

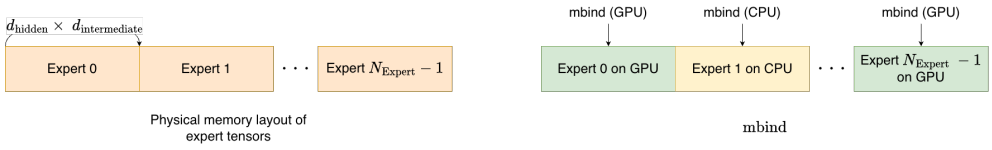


Figure 7: Memory layout of MoE expert weights in a `torch.tensor` (left) and how `mbind` controls their physical placement across GPU and CPU memory (right).

```
torch::Tensor make_heterogeneous_expert_tensor(
    const std::vector<int64_t>& sizes,
    torch::ScalarType dtype,
    torch::Device device,
    std::vector<bool> on_gpus,
    bool requires_grad
){
    // Compute the total tensor size in bytes.
    int64_t numel = 1;
    for (auto s : sizes)
    numel *= s;

    size_t elem_size = c10::elementSize(dtype);
    size_t bytes = numel * elem_size;

    // Allocate the tensor as system memory.
    void* ptr = mmap(nullptr, bytes,
                     PROT_READ | PROT_WRITE,
                     MAP_PRIVATE | MAP_ANONYMOUS,
                     -1, 0);

    // Request transparent huge pages for this region.
    if (madvise(ptr, bytes, MADV_HUGEPAGE) != 0) {
        perror("madvise");
    }

    // register a pointer (these functions are implemented in PyTorch)
    extension::registerNUMAPointer(ptr);
    auto deleter = [bytes](void* p) {
        extension::unregisterNUMAPointer(p);
        munmap(p, bytes);
    };

    /* ... */
    /* Build strides, tensor options and
    initialize a NUMA cpumask and gpumask used for mbind */

    size_t total_size = 0;

    // Assign each expert to either the CPU or GPU NUMA node.
    for (size_t i = 0; i < on_gpus.size(); i++) {

        // Each expert occupies one contiguous memory region.
        size_t start = i * strides[0] * elem_size;
        size_t size  = strides[0] * elem_size;

        // Select the NUMA node according to the target residency.
        unsigned long* mask =
```

```
            on_gpus[i] ? gpumask : cpumask;

        char* base = static_cast<char*>(ptr);
        void* p = base + start;

        // Bind this expert's memory region to the selected node.
        if (mbind(p, size, MPOL_BIND,
                  mask, maxnode, 0) != 0) {
            perror("mbind");
        }

        total_size += size;
    }

    TORCH_CHECK(total_size == bytes,
                "invalid byte calculations");

    // Touch the huge pages to materialize the allocated memory.
    memset(ptr, 1, bytes);

    // Wrap the system memory as a PyTorch tensor.
    return torch::from_blob(
        ptr,
        sizes,
        strides,
        deleter,
        options
    );

}
```

The resulting tensor is used to replace the storage referenced by the expert parameters of the original MoE model. Since the tensor retains the original expert dimension and memory layout, the existing grouped GEMM implementation can be reused without modification.

On GH200, CUDA Graphs can also operate on kernels accessing the `mmap`-allocated memory. Consequently, our implementation does not require the additional static buffers or CUDA Graph-specific memory management used by some existing expert offloading approaches, such as FreeToken. This simplifies the integration of expert offloading into existing inference frameworks such as vLLM.

A.1.3 Why should we choose NUMA APIs rather than CUDA Unified Memory APIs?

There are two main reasons for choosing the NUMA API. First, we found that huge pages could be effectively used with the NUMA API. In our experimental environment, system-allocated memory used a default page size of 64 KiB, while `cudaMalloc` used 2 MiB pages by default. This difference in page size affected the performance of the grouped GEMM kernel. Although `cudaMemAdvise` can be used to specify the preferred location of Unified Memory, we observed that specifying the CPU as the preferred location by `cudaMemAdvise` did not enable the use of huge pages. In contrast, using the NUMA `mbind` interface to bind the memory to the CPU NUMA node enabled huge pages to be used. We therefore adopted the NUMA API for our memory placement mechanism.

Second, the NUMA API provides explicit control over memory placement and allows us to avoid unintended page migration. On GH200, the Unified Memory subsystem can automatically migrate frequently accessed pages toward the GPU based on access counters (Schieffer et al., 2024). We observed that this behavior could cause data in CPU memory to be implicitly migrated to GPU memory, resulting in unintended performance gains and the increased GPU memory footprint. By using `mbind`, we could explicitly bind the memory to the CPU NUMA node and prevent such unintended migration. The relevant access-counter mechanism is exposed through

`/sys/module/nvidia_uvm/parameters/uvm_perf_access_counter_threshold`. Thus, we use the NUMA API to explicitly control the placement of CPU-resident experts while retaining the ability to use huge pages and avoiding implicit migration by the Unified Memory subsystem. We note that future updates to the CUDA Unified Memory API may provide improved support for huge pages and explicit placement control, potentially making it a viable alternative.

A.1.4 A BRIEF GUIDE TO IMPLEMENTING THE METHOD IN VLLM

We implement our method in vLLM using its Plugin functionality. vLLM Plugins allow users to modify the initialization process before vLLM initializes components such as the model, scheduler, and KV cache. We use this functionality to replace the implementation of `UVAOffloader._maybe_offload_to_cpu`, which moves pointers to memory allocated by `cudaHostAlloc`. By replacing this method, we integrate our expert placement mechanism into vLLM without modifying the main vLLM codebase. The access frequency information used for static expert placement and the number of experts to be placed in CPU memory are passed through environment variables. We then achieve expert-wise offloading by using the heterogeneous tensor construction function described in the previous section A.1.2.

A.2 HARDWARE

During the experiments, we additionally used an H100 server alongside the GH200 because some of the compared methods support only x86-64 systems. The H100 NVL and the GH200 both use HBM3 memory, and their GPU memory bandwidths are similar (3.9 TB/s and 4.0 TB/s, respectively). Therefore, the primary hardware differences relevant to our experiments are the CPU and the CPU-GPU interconnect. The details are shown in Table 1.

Table 1: Hardware configurations used in our experiments.

System	**GPU (VRAM, BW)**	**Interconnect (Bandwidth)**	**CPU (Arch, DRAM, BW)**
GH200	NVIDIA GH200 (96 GB HBM3, 4 TB/s)	NVLink-C2C (900 GB/s bidirectional)	Grace CPU (Arm, 480 GB LPDDR5X, up to 384 GB/s)
x86-64 + PCIe + H100	NVIDIA H100 NVL (94 GB HBM3, 3.9 TB/s)	PCIe Gen5 (128 GB/s)	AMD EPYC 9V84 (x86-64, 320 GB DDR5, 460.8 GB/s)