Memory Hierarchy

and Offloading for LLM Inference

February 2026

Table of Contents

1. Introduction

When model weights, KV-cache, and activations collectively exceed the available GPU memory—even after quantization and distribution across multiple chips—the system must leverage a deeper memory hierarchy. Modern servers offer a cascade of storage tiers with decreasing bandwidth and increasing capacity: GPU HBM/GDDR (1–3 TB/s), CPU DRAM (50–200 GB/s), and NVMe SSDs (3–7 GB/s for consumer drives, up to 14 GB/s for enterprise). Offloading strategically places data across these tiers to enable inference of models that would otherwise not fit, at the cost of increased latency unless carefully managed.

This report examines the memory hierarchy relevant to LLM inference, the principles of effective offloading, and the systems and techniques that make heterogeneous-memory inference practical on small-GPU deployments.

2. The Memory Hierarchy

2.1 GPU Memory (HBM / GDDR)

GPU memory is the fastest tier and the most constrained. Consumer GPUs typically offer 8–24 GB of GDDR6/GDDR6X with bandwidths of 500–900 GB/s. Data center GPUs provide 24–80 GB of HBM2e/HBM3 at 1.5–3.3 TB/s. For the decode phase of LLM inference, which is memory-bandwidth-bound, GPU memory bandwidth directly determines token generation speed. The goal is to keep all data accessed on the critical path—the current layer's weights and the active KV-cache—resident in GPU memory.

2.2 CPU Main Memory (DRAM)

Modern servers provide 128 GB to 2 TB of DDR4/DDR5 DRAM with aggregate bandwidth of 50–200 GB/s (depending on channel count). DRAM is roughly 10–30x slower than GPU HBM but 5–80x larger. It serves as the primary overflow tier for weights that don't fit in GPU memory. Data transfer between CPU and GPU occurs over PCIe, which limits practical throughput to 25–32 GB/s (PCIe 4.0 x16) or 50–64 GB/s (PCIe 5.0 x16).

2.3 NVMe Storage

NVMe SSDs offer terabytes of capacity at 3–7 GB/s sequential read bandwidth. While far too slow for real-time token generation, NVMe serves as a backing store for weight prefetching and for persisting KV-cache for session resumption. With careful scheduling, NVMe reads can be overlapped with GPU computation on other layers, partially hiding the latency.

2.4 The Bandwidth Cliff

Memory Tier	Typical Capacity	Bandwidth	Relative Speed
GPU HBM3	24–80 GB	2,000–3,300 GB/s	1x (baseline)
GPU GDDR6X	8–24 GB	500–900 GB/s	0.2–0.4x

CPU DDR5	128 GB–2 TB	50–200 GB/s	0.02–0.08x
PCIe 4.0 x16	(transfer link)	25–32 GB/s	0.01–0.013x
NVMe SSD	1–8 TB	3–7 GB/s	0.001–0.003x

The table illustrates the steep bandwidth cliff between tiers. Moving even a fraction of the critical-path data off-GPU can dramatically impact per-token latency.

3. Offloading Strategies

3.1 Weight Offloading

The simplest offloading strategy keeps some transformer layers' weights in CPU RAM and transfers them to GPU just before they are needed. Since a transformer processes layers sequentially, the system can prefetch layer N+1's weights from CPU while computing layer N on GPU. If the prefetch transfer time is less than or equal to the compute time for one layer, the PCIe transfer is fully hidden and the offloading is essentially free in terms of latency.

In practice, for small models or quantized weights, the per-layer compute time may be shorter than the transfer time, creating a bottleneck. The key parameters are: weight size per layer (model parameters / num layers, in bytes at the chosen precision), PCIe bandwidth, and per-layer compute time. When the transfer is slower, double-buffering (maintaining two weight buffers on GPU and alternating between them) ensures the GPU is never stalled waiting for a transfer to complete, though it does reduce the effective GPU memory available for KV-cache.

3.2 KV-Cache Offloading

For long-context inference or high-batch-size serving, the KV-cache can exceed GPU memory even when all weights fit. KV-cache offloading moves the cache for inactive or lower-priority sequences to CPU RAM, bringing it back when needed. This is naturally compatible with PagedAttention's block-based memory management: individual pages can be migrated between GPU and CPU without disrupting the overall cache structure.

A more sophisticated approach offloads KV-cache to CPU asynchronously during token generation and prefetches it when a sequence is scheduled for its next decode step. This requires the serving scheduler to have advance knowledge of which sequences will be active next—which continuous batching systems like vLLM naturally provide.

3.3 Hybrid CPU-GPU Inference

Rather than offloading data that the GPU will eventually process, hybrid inference assigns some layers to execute directly on the CPU. This eliminates the transfer overhead entirely for those layers, at the cost of slower CPU compute. The approach works best when the GPU handles the majority of layers and the CPU handles a small number of overflow layers. llama.cpp's GPU layer splitting is the most widely used implementation: the user specifies how many layers to place on each GPU, and remaining layers run on CPU.

3.4 FlexGen and Throughput-Oriented Offloading

FlexGen demonstrated that with careful scheduling across the GPU-CPU-SSD hierarchy, offline (non-interactive) inference can achieve high throughput even when the model is much larger than GPU memory. The key insight is to process a large batch of sequences through one layer at a time, maximizing the compute-to-transfer ratio. FlexGen's linear programming-based scheduler automatically determines the optimal data placement and batch ordering given the specific hardware's bandwidth and capacity constraints.

While FlexGen's approach introduces latencies of seconds to minutes per token (making it unsuitable for interactive chat), it is valuable for batch workloads like evaluation, data processing, and offline content generation where throughput matters more than latency.

4. Advanced Techniques

4.1 Compression During Transfer

Compressing data before transferring it over the PCIe bus can effectively increase the link's throughput. Weight matrices can be quantized to lower precision for transfer (e.g., from FP16 to INT4) and dequantized on the GPU after arrival, using the GPU's compute to decompress faster than the extra data could be transferred. KV-cache blocks can similarly be compressed before offloading to CPU and decompressed on retrieval. The overhead of compression/decompression is typically small relative to the transfer time savings.

4.2 Prefetching and Scheduling

The effectiveness of offloading depends almost entirely on whether transfers can be overlapped with computation. This requires explicit scheduling of asynchronous DMA transfers on dedicated copy engines (CUDA streams) that operate concurrently with compute kernels. The scheduling must ensure that data arrives in GPU memory before the compute kernel needs it, which requires lookahead of at least one layer (for weight prefetching) or one scheduling round (for KV-cache prefetching).

4.3 Unified Memory and Managed Memory

CUDA's Unified Memory (UM) provides a single virtual address space spanning GPU and CPU memory, with automatic page migration handled by the driver. While conceptually simple, UM's performance for LLM inference is generally poor because the page fault and migration overhead is unpredictable and not aligned with the model's access patterns. Explicit offloading with manual scheduling almost always outperforms UM for this workload. However, UM can be useful for prototyping and for handling rare edge cases (e.g., exceptionally long sequences that occasionally overflow GPU KV-cache).

5. Practical Recommendations for Small-GPU Systems

- **Quantize first, offload second:** Always apply weight quantization (INT4/INT8) before resorting to offloading. A 70B model at INT4 (35 GB) might fit across two 24 GB GPUs with no offloading needed.
- **Profile the bandwidth budget:** Measure actual PCIe throughput (not theoretical) using tools like bandwidthTest from CUDA samples. Real throughput is often 20–30% below spec due to protocol overhead.
- **Use llama.cpp for simple setups:** Its GPU layer splitting is battle-tested, easy to configure, and handles the double-buffering and transfer scheduling automatically.
- **Reserve GPU memory for KV-cache:** When planning offloading, prioritize keeping the KV-cache on GPU over keeping all weights on GPU. The KV-cache is accessed repeatedly per token, while each weight layer is accessed only once per forward pass.
- **Consider NVMe only for batch workloads:** The bandwidth cliff from DRAM to SSD is too steep for interactive inference. Use NVMe for offline processing or KV-cache persistence across sessions.
- **Monitor PCIe contention:** In multi-GPU systems, multiple GPUs may share PCIe lanes through a switch, reducing per-GPU bandwidth. Check the topology with nvidia-smi topo -m and account for shared bandwidth in scheduling.

6. Conclusion

Memory offloading extends the effective memory capacity of small-GPU systems, enabling inference of models that would otherwise be inaccessible. The trade-off is increased complexity and, unless transfers are fully overlapped, increased latency. For interactive workloads, the most practical approach is to combine quantization (to minimize what needs to fit in memory) with a modest amount of CPU offloading (a few layers) using double-buffered prefetching. For batch workloads, more aggressive offloading across the full GPU-CPU-SSD hierarchy can unlock throughput-efficient inference of very large models on very modest hardware. The key principle throughout is to measure the actual bandwidth at each tier and schedule transfers to hide behind computation wherever possible.