Summarize the article

Summary of arXiv:2503.18292 – "JENGA: Effective Memory Management for Serving LLM with Heterogeneity"

This paper presents **JENGA**, a novel memory management framework designed for efficiently serving large language models (LLMs) with heterogeneous architectures. As LLMs become more complex—incorporating features like varying embedding sizes, multimodal inputs (e.g., vision-language models), and diverse attention mechanisms (such as sliding windows and state-space layers like Mamba)—the traditional memory management solutions, like PagedAttention, are no longer optimal.

Key Points:

- **Problem Motivation:**
    - Serving LLMs is GPU-memory-intensive because every token's embedding for all prefixes in a request batch must be cached in GPU memory.
    - Modern LLMs introduce diversity in embedding sizes (e.g., mixing vision and text) and token dependency patterns, resulting in inefficiencies and fragmentation under current memory managers like PagedAttention.
- **What JENGA solves:**
    1. **Minimizes Memory Fragmentation:** Handles different-sized embeddings efficiently using a two-level allocator, with the least common multiple (LCM) of the different embedding sizes as the base page size.
    2. **Custom Caching & Eviction Policies:** Offers APIs to tailor caching/eviction to the precise token dependencies of each attention layer type (e.g., sliding window attention, Mamba, cross-attention, vision encoders).
- **Technical Approach:**
    - **Two-level Allocator:** Allocates large pages sized to the LCM of all embedding sizes and subdivides them into small pages matched to each layer's needs, greatly reducing wasted memory.
    - **Request-aware Allocation:** Allocates groups of small pages to the same request, ensuring pages can be efficiently freed when a request finishes.
    - **Customizable Prefix Caching:** Lets each layer type define its cache "hit" and eviction logic, aligned for maximum cache reuse without sacrificing performance.
- **Results and Impact:**
    - Integrated into **vLLM** (a leading LLM inference engine), JENGA:

        - **Improves GPU memory utilization by up to 79.6%.**
        - **Boosts serving throughput by up to 4.92× (1.8× on average).**
        - **Handles larger batches and more complex models without increasing latency.**
        - **Reduces memory waste from fragmentation in multi-modal and multi-attention LLMs.**
- **Real-World Applicability and Evaluation:**
    - Supports a variety of real models (e.g., Llama vision variants, Gemma-2, Ministral, Jamba).
    - Thorough benchmarks show that JENGA's design is robust across model types, with significant gains on both H100 and L4 GPUs.

Conclusion:
JENGA addresses the new complexities introduced by heterogeneity in modern LLMs (varying embedding sizes and attention mechanisms). By rethinking memory allocation and providing fine-grained control over caching and eviction per layer type, it significantly increases inference throughput and GPU utilization for state-of-the-art LLM services without requiring kernel changes—all while retaining backward compatibility with existing inference systems.

⁂

recent LLMs are more heterogeous? what changed?

Yes, **recent large language models (LLMs) have become much more heterogeneous** compared to earlier generations. Here's what has changed:

- **Varied Embedding Sizes:** Earlier LLMs used a single type (text), but modern models often fuse vision and text, requiring different token embedding sizes for each input (multimodal models).[2]
- **Diverse Attention Mechanisms:** New models include a mix of self-attention, sliding window attention, grouped attention, and emerging alternatives like state-space layers (e.g., Mamba). Each has unique memory and computation patterns.[2]
- **Layer Diversity:** Instead of stacking identical transformer blocks, new models may interleave different types (vision encoder blocks, cross-attention layers, mixture-of-experts, etc.) within one model.
- **Token Dependency Patterns:** Some models focus on local context (shorter sliding windows) or stateful mechanisms (like Mamba) rather than global-context self-attention.[2]

Key Differences from Earlier LLMs:

Aspect	Old LLMs (e.g., GPT-3)	Recent LLMs (e.g., Llama Vision, Jamba)
Embedding Types	Text-only, fixed size	Text + vision, mixed sizes
Attention Layers	Self-attention only	Sliding window, mixture, state-space

Aspect	Old LLMs (e.g., GPT-3)	Recent LLMs (e.g., Llama Vision, Jamba)
Layer Homogeneity	Identical transformer blocks	Heterogeneous layers
Memory Management	Uniform and simple	Requires custom, flexible strategies

Why This Matters:
This heterogeneity enables modern LLMs to:

- Process multimodal data (e.g., images+text)
- Scale more efficiently for longer context windows
- Implement new reasoning and memory mechanisms

But it also makes efficient serving and inference optimization much harder, motivating memory/caching advances like JENGA.[2]

Summary and insights from "JENGA: Effective Memory Management for Serving LLM with Heterogeneity" (arXiv:2503.18292).[2]

⁂

what is the single state token?

In recent LLM architectures with *state-space layers* such as Mamba, **the "single state token" refers to the idea that only the latest token's state (i.e., a single, large tensor) must be stored at each decoding step**, rather than keeping all previous tokens' states as in standard self-attention.

- In classic self-attention, every new token depends on all previous tokens—the system stores key-value (KV) cache for all tokens in the prefix.
- In Mamba-style state-space layers, *the model only needs the most recent token's state tensor*—not the full prefix—because the context is integrated into a single, recurrent state that gets updated at each step.
- This "single state token" is much larger than a regular KV-cache entry, but it massively reduces the number of cached items—just one per layer, not one per token.

Why is this important?

- It minimizes memory usage for these layers.
- Memory managers (like JENGA) must handle this high-variance: small KV-caches for attentions and large single-token state caches for layers like Mamba, sometimes interleaved within one model.

Summary:
The "single state token" is the large, persistent per-layer tensor used by state-space layers (e.g., Mamba). It only requires the last token's state to be stored, not all previous tokens. This design underpins key innovations in modern heterogeneous LLMs.[3]

See Figure 1 and §3.1 ("State space models...a large but fixed-size tensor to capture the context...updated recurrently...only need for the last token") in the JENGA paper.[3]

⁂

what is the window layer?

A **window layer** in the context of recent LLM architectures refers to a transformer layer that uses **window-based (local) attention** rather than full self-attention.

Key Characteristics:

- Instead of attending to all previous tokens (global attention), a window layer **restricts attention to a fixed-size window of recent tokens**—for instance, only the last 128 tokens.
- This reduces both computation and memory costs, allowing models to efficiently handle much longer sequences.
- The output at each position can only depend on a small, recent window of context.

Why are window layers used?

- For long sequences, full attention mechanisms scale poorly, since memory and compute grow quadratically with input length.
- Window layers maintain efficiency by focusing only on the most relevant, recent context, which is often sufficient for many tasks, especially in language modeling.

Implications for memory management:

- Caching in window layers needs to keep only the tokens within the window, not the whole sequence.
- Modern systems (like JENGA) support custom cache management for these layers—**evicting tokens outside the window** to save memory.

Summary:
A window layer is any transformer-like layer that performs attention within a fixed sliding window over the recent sequence, rather than over the entire sequence. This approach is common in efficient long-context LLMs.[4]

See JENGA paper §3.1, Figure 1, and the discussion of "window attention layer" and prefix cache strategies for window-based dependency.[4]

⁂

explain the jenga architecture

The **JENGA architecture** is designed to efficiently manage GPU memory for serving modern, heterogeneous LLMs—especially those that use a mix of embedding sizes and layer types. Here’s an explanation of its core design:

1. Two-Level Memory Allocator

- **Large pages (base page):**
  JENGA's key innovation is using a *two-level allocation strategy*. The system first allocates **large pages** of GPU memory, with each large page sized to the *least common multiple (LCM)* of all possible embedding sizes in the model. This prevents waste caused by varied embedding sizes in different layers (e.g., text, vision).
- **Small pages:**
  Each large page is then partitioned into **small pages** of sizes that match the needs of each specific layer (such as different attention or state layers). This allows precise memory fitting for each use-case and reduces fragmentation.

2. Request-Aware Page Grouping

- **Grouped allocation:**
  When a user request is handled, the necessary small pages for all layers and all tokens in that request are allocated together as a *group*. Once the request finishes, all its pages can be released together, minimizing memory leaks and fragmentation.

3. Customizable Caching and Eviction Policies

- **Layer-specific cache logic:**
  Each layer (self-attention, window attention, state-space) can define its own caching and eviction policy, controlling what tokens or states must be kept:
    - **Self-attention:** Keep all past tokens' cache.
    - **Window attention:** Keep only recent tokens within the "window," evict older ones.
    - **State-space/Mamba:** Only keep the latest state token.
- JENGA exposes APIs so each layer can signal which past states are required and when memory can be freed.

4. Minimal Overhead and Compatibility

- **No kernel changes:**
  JENGA is implemented as a user-space memory manager. It requires no changes to GPU kernels or frameworks, so it can fit seamlessly into inference engines like vLLM.
- **Backwards-compatible:**
  It is fully interoperable with traditional (homogeneous) LLMs as well as new heterogeneous ones, supporting mixed workloads effortlessly.

Visualization/Analogy:
Imagine JENGA as a clever packing system:

- It *first* grabs a box just big enough to hold items of all possible sizes (the LCM trick),
- *Then* it neatly divides the box into compartments matched exactly to each item.
- When you're done with an order, the whole box (and all compartments) gets cleared.

- And each “customer” (layer) can specify which compartments it needs to keep—or toss—at any time.

In short:
JENGA’s architecture ensures efficient, flexible, and fragmentation-free GPU memory allocation for today’s complex, multi-modal, mixed-layer LLMs. This leads to much higher throughput and resource utilization when serving state-of-the-art models.[5]

See Figure 3 and Sections 4–5 in the JENGA paper for detailed technical diagrams and system workflow.[5]

⁂

how much changes is needed to LLM model?

Very little change is required to LLM models to use JENGA. The key advantage of JENGA is that it is implemented as a **user-space memory manager** and provides a **simple API for cache and eviction logic**. This means:

- **No changes to model weights or architecture:**
  The LLM itself and its layers do not need to be rewritten or retrained.
- **No changes to GPU kernels or inference engine fundamentals:**
  JENGA integrates with inference frameworks (like vLLM) without altering core GPU operations or the way layers are executed.
- **Minimal code changes:**
  Some minor changes are needed to plug JENGA’s prefix cache APIs into the LLM server code. Specifically:
    - Each layer type defines (via JENGA’s API) its cache usage pattern (i.e., which tokens it needs to cache or evict).
    - The inference server calls JENGA’s allocator for prefix caching rather than hard-coded policies.
- **Backward compatibility:**
  You can still serve traditional LLMs (with uniform layers and embedding sizes) without modification.

Summary:
You **do not need to modify the model itself**—just adapt the handling of prefix caches and token dependencies in your inference server or runtime, using JENGA’s interface. This enables all the memory optimization benefits for heterogeneous models with almost no disruption to the modeling workflow.[6]

See JENGA paper, §4 ("Model Compatibility") and §6 ("Implementation").[6]

⁂

what is the downside of GCD?

The main **downside of using the GCD (greatest common divisor) for page sizing in memory allocation**—as opposed to the LCM (least common multiple), which is JENGA’s choice—is **increased memory fragmentation and inefficient use of GPU memory** when dealing with layers that require varied embedding sizes.

Why is GCD problematic?

- **Wasted Memory:**
  If you divide memory into small blocks sized by the GCD of all embedding sizes, you may have many small blocks left over that can't be efficiently used by layers needing bigger blocks. These leftovers accumulate and lead to fragmentation.
- **Cannot Fit Larger Embeddings:**
  Layers with large embedding sizes would need to combine multiple GCD-sized blocks, but unless the allocation fits perfectly, you'll waste memory at the boundaries (not enough blocks to fully pack the larger layer’s needs).
- **Complex Allocation Overhead:**
  Managing many tiny blocks (GCD-sized) increases allocation overhead and fragmentation, making caching and eviction for variable-size needs far less efficient.

JENGA’s Solution (LCM):

- Allocating pages at the LCM of all embedding sizes ensures that every embedding size can fit precisely into the allocation with zero leftover space for any layer.
- This reduces fragmentation and improves overall memory utilization.

Summary:
Using GCD for page sizing leads to poor packing, leftover memory, and fragmentation. **LCM is preferred for heterogenous layers** because it enables perfect fitting for all relevant embedding sizes, minimizing waste and maximizing throughput for variable-size workloads.[7]

See JENGA paper §4 ("Design") discussion of page sizing, and the comparison between GCD and LCM strategies for minimizing fragmentation.[7]

⁂

explain why is LCM better?

LCM (Least Common Multiple) is better for page sizing in heterogeneous LLM memory management because it ensures every possible embedding size can fit into the page allocation without wasted space.

Here’s why:

- **Perfect Fit for All Embedding Sizes:**
  The LCM of all possible embedding sizes is the smallest number such that every embedding size divides it with no remainder. This means:

    - Any layer, no matter its embedding size, can be assigned a portion of the base page allocation that fits its needs *exactly*—with no leftover fragments.
- **Zero Fragmentation:**
  Unlike small GCD-sized blocks (which can leave unusable slivers when allocating to large embeddings), an LCM-sized page can always be subdivided into allocations for all layer types, whether they use small text, large vision, or other custom embedding sizes.
- **Simpler, More Efficient Allocation:**
  Managing a pool of LCM-sized pages keeps the allocator logic simple and ensures high packing efficiency, instead of juggling many small sizes and risking "holes" in GPU memory.

Example:

- Suppose you have two embedding sizes: 256 and 384.
- GCD = 128: allocating in chunks of 128 leaves leftovers for 384, which is not a multiple of 256.
- LCM = 768: allocating in chunks of 768 lets you fit exactly three 256-sized allocations or two 384-sized ones, with no waste.

In summary:
LCM-based page sizing eliminates internal fragmentation and fits all heterogeneous layer needs perfectly, leading to **higher GPU utilization, reduced waste, and improved throughput** for serving modern LLMs with diverse architectures.[8]

As detailed in JENGA, §4 ("PagedAllocator with Heterogeneous Embeddings").[8]

⁂

1. https://arxiv.org/pdf/2503.18292.pdf
2. https://arxiv.org/pdf/2503.18292.pdf
3. https://arxiv.org/pdf/2503.18292.pdf
4. https://arxiv.org/pdf/2503.18292.pdf
5. https://arxiv.org/pdf/2503.18292.pdf
6. https://arxiv.org/pdf/2503.18292.pdf
7. https://arxiv.org/pdf/2503.18292.pdf
8. https://arxiv.org/pdf/2503.18292.pdf