explain this paper to me

Here's a concise explanation of the paper **"Efficient Memory Management for Large Language Model Serving with PagedAttention" (vLLM)**:

- **Problem:**
  - Serving large language models (LLMs) like GPT or LLaMA requires high throughput, but is bottlenecked by GPU memory usage—mainly for storing key-value (KV) caches used in attention.
  - Existing LLM serving systems allocate contiguous memory for each request's KV cache, causing huge waste due to fragmentation and inefficiency. Batch size (number of requests served simultaneously) and throughput are therefore limited.
- **Key Idea (PagedAttention):**
  - Inspired by classical virtual memory and paging concepts in operating systems, PagedAttention stores each request's KV cache in fixed-size blocks ("pages") that do not need to be physically contiguous.
  - This design eliminates internal/external fragmentation and allows flexible sharing and recycling of memory blocks, massively reducing memory waste.
- **System (vLLM):**
  - vLLM implements PagedAttention and a block-level memory manager.
  - The KV cache is managed like OS virtual memory: requests map their "logical" blocks to "physical" blocks in GPU memory, allowing dynamic growth/shrinkage and memory sharing across requests.
  - The system also supports swap-out to CPU and recomputation for blocks that exceed capacity.
- **Benefits:**
  - **Higher Throughput:** By reducing memory waste, vLLM fits more requests onto the same hardware, enabling batch sizes and throughput up to 2–4× that of leading alternatives (Orca, FasterTransformer), especially for longer sequences and complex decoding (beam search, parallel sampling).
  - **Zero memory waste:** Only one block per request is ever reserved/unused at a time, compared to massive overprovisioning in prior systems.
  - **Memory sharing:** Multiple outputs per request (e.g., from beam search or program assistants) can share prompt KV cache blocks, saving 6–66% of memory in practice.
  - **Flexible scheduling and distributed execution:** vLLM supports distributed GPU setups and fair scheduling, with its centralized block manager.

- **Takeaway:**
  - By adopting OS-inspired paging for LLM KV cache memory, vLLM presents a practical, highly optimized serving system for large models, enabling real-world deployment at higher efficiency, lower cost, and with broader support for advanced LLM applications.

If you'd like a deep dive into any part—like the algorithm, engineering implementation, or experimental results—I can explain those sections in detail!

⁂

how is KVCache stored?

KVCache storage in vLLM (with PagedAttention):

- **PagedAttention splits the KVCache into fixed-size blocks ("pages") rather than using one large, contiguous chunk of memory.**
  - Each block stores key and value vectors for a set number of tokens—this is analogous to memory paging in operating systems.
  - The blocks for a given sequence do **not need to be stored contiguously in GPU memory**.
- **Memory Management:**
  - **Logical blocks:** For each request, the cache is organized as a series of logical KV blocks, which are filled as new tokens are processed.
  - **Physical blocks:** The GPU allocates a buffer, divides it into physical KV blocks, and keeps a mapping table between logical and physical blocks (this "block table" says which actual GPU block holds which part of each sequence).
  - New physical blocks are allocated only as needed—no need to pre-reserve space for maximum possible sequence length, which drastically reduces memory waste.
- **Block Table & Translation:**
  - The block table records the mapping (logical → physical) and how full each logical block is.
  - This separation allows vLLM to **grow KVCache dynamically and efficiently**, limit fragmentation, and allow for memory sharing.
- **Physical Storage Features:**
  - KV blocks can be shared—for example, if multiple output sequences share a common prompt, only one copy of those shared tokens' KV blocks is needed.
  - When multiple outputs diverge, vLLM uses a "copy-on-write" approach: shared blocks are copied if modifications are needed, otherwise reference counts handle sharing.
- **Swapping & Recovery:**
  - If GPU memory runs out, vLLM can evict blocks to CPU RAM ("swapping") or recompute evicted blocks when needed.
  - These methods leverage the block-based structure to avoid waste.

Summary:
Compared to traditional LLM serving (which allocates a continuous tensor per request), vLLM stores the KVCache in small blocks/pages indexed by a table—allowing for non-contiguous memory use, sharing, dynamic allocation, and minimal waste.

If you want a simple diagram, think:

- Each sequence/request = row of logical blocks
- Logical blocks → linked to physical blocks (may be anywhere in GPU memory)
- Table tracks the mapping and sharing

Let me know if you want code-level or kernel-level details from the implementation!

⁂

how and why is the kvcache memory fragmented?

Fragmentation in KVCache Memory (Traditional Systems):

- **How fragmentation occurs:**
  - Most LLM serving frameworks allocate a **contiguous KVCache buffer per active sequence/request**.
  - Each buffer size is usually fixed for the *maximum possible sequence length* (needed for the longest output a user might request).
  - For most requests, actual output length is **much shorter than the reserved buffer**—so a large part of the buffer sits unused.
  - Different requests finish at different times and new requests start, leaving old, partially-filled buffers “stranded”—this is **internal fragmentation** (unused memory inside allocated buffers).
  - If trying to compact or merge buffers, the *contiguous* nature makes it hard—smaller unused fragments pile up (**external fragmentation**), resulting in even less usable memory for new requests.
- **Why this is a problem:**
  - With many requests, the sum of all unused but reserved memory fragments quickly eats up GPU capacity.
  - Even “dynamic” memory managers frequently run into fragmentation, as requests differ in output token counts, finish asynchronously, and require allocation/deallocation at different rates.
  - **Limits throughput**—the GPU cannot fit as many requests (batches), so overall system performance drops.

PagedAttention (vLLM) Solution:

- By splitting the KVCache into **blocks (pages)**, vLLM avoids the need for large, contiguous buffers and greatly reduces both types of fragmentation.

- Each page is reused, shared, and freed independently, so unused fragments are minimized.

In summary:

- *Fragmentation* happens in KVCache because each request typically gets a large buffer sized for worst-case output (for contiguous storage), but usually only needs a small part of it—leading to lots of wasted space and poor packing for high-throughput serving.
- *vLLM* solves this by using non-contiguous blocks and a mapping table, allowing near-zero waste and much higher throughput.

Let me know if you want more examples, formulas, or diagrams illustrating fragmentation!

⁂

what is the architecture of vLLM?

vLLM Architecture Overview:

- **Central Scheduler & Distributed Workers:**
  vLLM uses a *central scheduler* that coordinates multiple distributed GPU worker processes. The scheduler receives incoming requests and decides how to batch and schedule them for maximum throughput.
- **Paged KV Cache Manager:**
  The *KV cache manager* is the key innovation. It divides key-value cache memory into fixed-size blocks ("pages") instead of giving each request a single contiguous allocation. It maintains *block tables*—these map each request's logical KV blocks to available physical blocks in GPU memory (much like a virtual memory system in operating systems).
- **Block Allocation & Sharing:**
  Each GPU worker runs a *block allocator* for the memory pools (on both GPU and, for swapping, CPU). Through the scheduler, workers receive instructions on how to allocate, free, and transfer memory blocks as new tokens are generated or requests complete.
  - *Copy-on-write* and *reference counting* mechanisms track memory usage and enable sharing blocks across requests or outputs.
- **PagedAttention Kernel:**
  The *PagedAttention* computation kernel can fetch and use non-contiguous blocks efficiently while calculating attention over sequences. This flexibility supports efficient batching and memory usage.
- **Support for Complex Decoding & Distribution:**
  The mapping layer allows vLLM to handle various decoding algorithms (sampling, beam search, etc.), where memory sharing and dynamic allocation are crucial. In distributed execution, all GPU workers share the block tables through the scheduler, ensuring coherent and efficient multi-GPU support.
- **Frontend & Model Executors:**
  The *API server* and *inference engine* are built in Python (with custom CUDA kernels for

memory ops and attention). The API uses OpenAI-compatible endpoints for client requests, then vLLM coordinates the backend execution and memory management.

Summary Table:

Component	Purpose
Scheduler	Batches, schedules, and distributes inference requests
KV Cache Manager	Divides memory into blocks/pages; maps logical to physical blocks
Block Allocators	Manage memory pools on GPU/CPU; handle allocation, free, swap
Workers	Execute model inference; maintain block tables and work with cache
PagedAttention	Attention kernel with block-based memory access (not contiguous)
Frontend/API	Exposes endpoint for users (OpenAI API interface)

Key Benefits:

- Near-zero waste memory management
- Efficient batching, sharing, dynamic allocation
- Flexible support for decoding algorithms and distributed GPU setups

Let me know if you want a diagram or kernel-level details for any subcomponent!

⁂

in vllm, is the kvcache in GPU continuous? or paged?

In **vLLM**, the KVCache in GPU memory is **paged**, not continuous.

- Instead of allocating a single large, contiguous buffer for each sequence/request (as in traditional systems), vLLM's **PagedAttention** splits the KVCache into fixed-size blocks or "pages."
- These blocks/pages are **not stored contiguously** in GPU memory. They can be placed anywhere in the memory pool and tracked with a mapping table.
- This paged structure avoids fragmentation and allows sharing, dynamic allocation, and efficient use of GPU memory for multiple requests and complex decoding tasks.

Summary:
vLLM uses a paged KVCache on GPU—blocks are non-contiguous, managed with a table, and reused dynamically for efficiency. This is a key architectural advantage over traditional (continuous) allocation.

⁂

will the kernel code needs to be changed to adapt this system? can they still use traditional cublas libraries?

Yes, **kernel code must be changed to adopt vLLM's paged KVCache system**.

Why changes are needed:

- Traditional LLM serving relies on the **KVCache being stored contiguously in GPU memory**. Standard CUDA or cuBLAS-based attention kernels expect contiguous memory for fast batched computation.
- With **vLLM's paged design**, KV blocks for a given sequence are spread throughout GPU memory and accessed according to a mapping table (similar to virtual memory). The attention kernel must gather the correct blocks, which are **not guaranteed to be contiguous**.

Consequences:

- The attention computation (e.g., Multi-Head Self-Attention) now requires a **custom CUDA kernel** that can efficiently access these non-contiguous "pages," gather the keys and values for the active tokens, and perform attention calculations.
- **cuBLAS and standard CUDA kernels** for attention are designed for contiguous tensors and **cannot be directly reused** for PagedAttention without significant modification.
- vLLM provides its own optimized CUDA kernel for PagedAttention, which supports the block-based (paged) memory access pattern.

Summary Table:

Approach	KVCache Layout	Attention Kernel Needed	cuBLAS Usable?
Traditional LLM	Contiguous	cuBLAS/standard CUDA kernel	Yes
vLLM (Paged)	Paged (non-contig)	Custom PagedAttention CUDA kernel	No (for paged attention)

In essence:
vLLM requires specialized kernel code for its paged KVCache system. The traditional cuBLAS libraries and kernels cannot support this non-contiguous memory organization for the core attention operation; custom CUDA kernels are necessary and implemented as part of vLLM.

⁂

how large are each kv blocks?

Each **KV block** (or "page") in vLLM's PagedAttention typically stores the key and value tensors for **a fixed number of tokens**.

- The **page/block size** is a configurable parameter, often set to store **16 or 32 tokens per block**. This matches well with GPU memory alignment and kernel efficiency.

- The exact size (e.g., 16, 32, or 64 tokens per block) depends on hardware, model size, and tradeoff between memory reuse, fragmentation, and access efficiency.
- Each block contains the full key and value vectors for every token in that block, for all layers and heads.

Summary:
The typical KV block/page size in vLLM is **16 or 32 tokens per block, but configurable**. This allows efficient, low-fragmentation allocation and rapid block-level access for the attention kernel.

what is beam search?

Beam search is a popular **decoding algorithm** used in natural language generation (such as with large language models) to find high-probability output sequences (e.g., text completions, translations) efficiently.

How it works:

- At each step, instead of only keeping the single most likely next token (as in greedy search), beam search keeps track of the top **B best partial sequences** (where **B** is the *beam width*).
- For each candidate sequence in the current "beam," the algorithm examines all possible next tokens and computes their probabilities, then extends each sequence by one token with all possible choices.
- Out of all these extensions, it selects the **B highest-probability expanded sequences** for the next round.
- This process repeats step by step until completion (end-of-sequence token or max length).
- At the end, the highest-probability completed sequence among all beams is chosen as the final output.

Benefits:

- Balances between **exploration** (looking at multiple possible continuations) and **exploitation** (focusing on the most likely options) better than greedy decoding.
- Often yields substantially better quality outputs than greedy search, especially for tasks like translation and summarization, because it considers more possible sentence structures.

Example (beam width = 2):

- First token: keep the 2 highest-probability tokens.
- Next token: expand both, and keep the best 2 of all 2×vocab_size possible next tokens.
- Repeat until all candidates are complete or max tokens reached.

Tradeoffs:

- **Larger beam width** means more computation and memory use but generally better results.

- Beam search does not guarantee a globally optimal solution, but it balances quality and speed well in many applications.

If you'd like, I can also explain how vLLM's memory sharing reduces redundancy in beam search decoding!

⁂

what is the eviction policy for vllms?

vLLM's **eviction policy** for managing KVCache memory relies on a **sequence-level ("all-or-nothing") approach**. Here's how it works:

- **When GPU memory for KV blocks runs low:**
  vLLM selects entire *sequences* (not individual blocks) to evict. That means all blocks of a request/sequence are evicted together. [9]
- **Gang-scheduling and preemption:**
  If multiple sequences form a group (e.g., beam candidates from beam search), the whole group is preempted and evicted at once. This is because sequences in one group may share memory, so partial eviction isn't safe. [9]
- **Upon eviction, how does vLLM recover?**
  - **Swapping:** The evicted KV blocks (pages) are copied out from GPU to CPU memory. CPU memory acts as a swap space, bounded by the allocated size for KVCache in GPU.
  - **Recomputation:** If swapped blocks are needed again, vLLM can recompute them quickly, using the prompt phase to efficiently restore all needed KV values in a single pass if possible. [9]
- **Eviction Selection:**
  While typical OS paging systems use least-recently-used (LRU) or predictive policies, vLLM can choose sequences that arrived most recently for eviction (first-come-first-serve for fairness, newest requests evicted first).
  - No page-by-page eviction—only wholesale, per-sequence (or per-sequence-group) eviction. [9]

Summary:

- vLLM evicts the KV blocks for a whole sequence (or sequence group) together—*all or none*.
- Recovery uses swap-to-CPU or recomputation.
- This supports efficient memory sharing, avoids fragmentation, and fits LLM requirements better than fine-grained paging.

Let me know if you want more details on swapping versus recomputation tradeoffs!

⁂

recompute? are they not loaded back?

Great question! In vLLM, **evicted KV blocks can be restored in two ways: swapping or recomputation**.

- **Swapping:**
  When a sequence is evicted, its KV blocks are first copied to CPU memory ("swap space"). If those blocks are needed later (say, the same sequence is resumed for generation), vLLM can simply **copy the required blocks back from CPU RAM to GPU memory**—this is called swapping. This is fast if the data is still in CPU memory and the network/bus is quick.
- **Recomputation:**
  But if the evicted KV blocks are *not* available in CPU RAM (maybe swap space is limited or they never got swapped out), vLLM can **recompute** them. This means it reruns the LLM forward pass on the same prompt or prefix tokens to regenerate all necessary key and value tensors for that sequence.
  Recomputation is generally slower than a simple memory copy but is guaranteed to reconstruct the exact blocks, as long as the inputs and model weights are unchanged.

In summary:

- If KV data is still in CPU "swap," vLLM will load (copy) blocks back to GPU.
- If not (e.g., swap space overflow, or swap disabled), vLLM will **recompute** the missing KV blocks by forwarding the model over the same prompt tokens.

vLLM uses whichever option is available based on system status and performance tradeoffs. Both load (swap) and recompute are supported.

⁂

what is fastapi

FastAPI is a modern Python web framework for building **fast, efficient APIs**—especially RESTful APIs and web services.

Key features:

- **High performance:** Built on top of Starlette (for networking) and Pydantic (for data validation), FastAPI is designed to be as fast as Node.js or Go for high-load applications—thanks to asynchronous support.
- **Type hints:** You write API endpoints using **Python type annotations**, and FastAPI uses these to validate input/output data, generate documentation, and enable auto-completion in editors.
- **Automatic documentation:** With every API you create, FastAPI automatically generates interactive OpenAPI (Swagger) docs for your endpoints. You can test them in the browser with no extra setup.
- **Easy to use:** Very concise, yet flexible—great for both prototypes and large production services.

- **Built-in data validation:** Uses Pydantic models for parsing and checking request data.
- **Asynchronous execution:** Supports async/await, making it suitable for IO-heavy workloads and maximizing throughput.

Common uses:

- Creating endpoints to serve ML models or LLMs (including vLLM!).
- Building data services, microservices, and backend REST APIs.
- Rapid prototyping and production-level deployment.

Example FastAPI endpoint:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/hello")
def say_hello(name: str):
    return {"message": f"Hello, {name}!"}
```

This creates an HTTP GET endpoint at `/hello?name=...` that returns a greeting.

In summary:
FastAPI is a lightweight, high-performance Python framework for building APIs quickly, with excellent type safety, validation, and automatic docs generation. It's widely used in ML/deep learning serving stacks, including vLLM, for deploying models and services.

⁂

for contiguous memory, the kernels can benefit on memory bursts, how do they handle that with such page?

Excellent point! **Contiguous memory** enables kernels (like those using cuBLAS) to exploit "memory burst" (efficient coalesced reads/writes) for high throughput.
With vLLM's **paged (non-contiguous) KVCache structure**, maintaining such efficiency is a challenge.

How vLLM handles this:

- **Custom CUDA Kernels:**
  vLLM implements a specialized **PagedAttention CUDA kernel**. Instead of relying on contiguous memory access, this kernel:
  - Takes a mapping table as input, which tells it where each block/page for each sequence is located in GPU memory.
  - Gathers keys and values from these scattered pages into shared memory or thread-local buffers **before** running the attention computation.

- **Batching Blocks for Efficiency:**
  The kernel batches the access patterns as much as possible, minimizing random reads and ordering block fetches to take advantage of GPU memory bandwidth when possible.
- **Prefetch & Coalescing:**
  If multiple sequences share pages (for example, common prompt tokens across beam search candidates), the kernel recognizes this and orders access to maximize burst reads for shared blocks.
- **Memory Alignment:**
  Blocks (pages) are sized to match GPU memory alignment requirements, so individual block copies can still be performed efficiently even though the overall sequence is non-contiguous.
- **Latency vs. Throughput:**
  While some raw burst efficiency from contiguous memory is inevitably lost, the **overall throughput actually increases** due to reduced fragmentation—more requests fit in GPU RAM, more batches run at once. The design sacrifices some burst efficiency for much higher batch-level efficiency and scalable serving.

Summary:

- vLLM's kernels use mapping tables and efficient gather/copy strategies to process paged KVCache, partially compensating for the lack of contiguous-burst benefits.
- Block size/alignment and shared-access optimization further enhance throughput.
- The tradeoff is a small loss in burst speed but a major gain in overall model serving efficiency due to better memory packing and less waste.

If you want pseudocode or details of how blocks are gathered and computed in the kernel, let me know!

⁂

what is the TTFT (time to first token) impact of vllm?

vLLM's impact on TTFT (Time To First Token):

- vLLM's primary benefit is **higher throughput**—it fits more requests per batch, thanks to efficient KV cache memory handling—but it is specifically designed to keep **latency (including TTFT)** similar to or only slightly higher than other systems.
- According to benchmarks, **vLLM maintains about the same level of latency/TTFT per token as Orca and FasterTransformer, but** can serve many more requests in parallel before latency increases.
- There is a small kernel-level overhead (~20–26% higher attention kernel latency) due to block mapping, but this is limited to the attention step and doesn't substantially affect the overall TTFT or end-to-end latency. All other compute operators (linear layers, etc.) stay fast.[13]

- As batch size and load increase, TTFT remains low for much higher request rates compared to alternatives. Throughput increases by 2–4× with the same latency and TTFT profile.
- For “cold start” or initial token generation, the impact is minimal—most performance gains are on larger batches and higher load.

Summary:

- **vLLM’s TTFT is comparable to state-of-the-art systems at low/moderate loads.**
- At high throughput, vLLM keeps TTFT low (doesn’t degrade sharply), while other systems see TTFT and overall latency spike sooner.
- The overall user experience: equal or slightly better TTFT under real-world loads, with dramatically more requests served at once.

⁂

1. https://arxiv.org/pdf/2309.06180.pdf
2. https://arxiv.org/pdf/2309.06180.pdf
3. https://arxiv.org/pdf/2309.06180.pdf
4. https://arxiv.org/pdf/2309.06180.pdf
5. https://arxiv.org/pdf/2309.06180.pdf
6. https://arxiv.org/pdf/2309.06180.pdf
7. https://arxiv.org/pdf/2309.06180.pdf
8. https://arxiv.org/pdf/2309.06180.pdf
9. https://arxiv.org/pdf/2309.06180.pdf
10. https://arxiv.org/pdf/2309.06180.pdf
11. https://arxiv.org/pdf/2309.06180.pdf
12. https://arxiv.org/pdf/2309.06180.pdf
13. https://arxiv.org/pdf/2309.06180.pdf