

vLLM

Efficient Memory Management for
LLM Serving with Paged Attention

What are Large Language Models? (LLMs)

- Large Language Models (**LLMs**) are **neural networks trained on vast text data** to generate, summarize, and understand human language.
- They are **based on transformer architectures**, enabling them to process and generate coherent text.

Key Characteristics:

- **Autoregressive token generation** – predicts the next word given prior context.
- **Trained on massive datasets** – books, articles, web data.
- **Capable of zero-shot and few-shot learning** – can generalize to new tasks with minimal examples.
- **Example models:** GPT-4, LLaMA, PaLM, Mistral.

What is LLM Inference?

- **LLM inference** is the process of **running a trained model** to generate text, embeddings, or responses based on a given prompt.

How It Works:

1. **Input Prompt:** The user provides a sequence of tokens (text input).
2. **Autoregressive Generation:** The model predicts the **next token** based on prior tokens.
3. **Token-by-Token Output:** The process continues until a termination condition is met (e.g., end of sentence, max token limit).

Key Factors in Inference:

- **Latency-sensitive:** Real-time applications require fast response times.
- **Memory-intensive:** Needs efficient handling of the Key-Value (KV) cache.
- **Compute-heavy:** Requires GPUs or TPUs for fast execution.

Why Are GPUs Used for LLM Training & Inference?

Parallel Processing Power

- ✓ **GPUs handle thousands of operations in parallel**, unlike CPUs which process tasks sequentially.
- ✓ **Essential for LLMs** that require massive matrix multiplications (e.g., self-attention layers in transformers).

Feature	GPUs	CPUs
Parallelism	Thousands of cores for matrix math	Few cores optimized for sequential tasks
Memory Bandwidth	High (fast access to model parameters)	Lower (slower for large models)
Efficiency	Designed for deep learning workloads	General-purpose computing
Speed for AI	10×–100× faster	Much slower for LLMs

How Does LLM Inference Differ from Model Training?

Aspect	Model Training	Model Inference
Purpose	Teaches the model to predict next tokens	Uses trained model to generate new tokens
Process	Forward + backward pass, optimizing weights	Only forward pass, computing token probabilities
Compute Cost	High (uses multiple GPUs for long periods)	Lower, but still expensive for large models
Memory Usage	Requires storing gradients & optimizer states	Primarily uses KV cache for previous tokens
Time Scale	Takes weeks/months to train a model	Runs in milliseconds to seconds per query

What Are the Key Challenges in Serving Large Language Models?

1. Memory Bottlenecks

- **KV cache growth:** Each request requires storing **attention key-value pairs**, consuming large GPU memory.
- **Memory fragmentation:** Inefficient allocation reduces usable space.

2. Latency vs. Throughput Trade-off

- **Low-latency** needed for chatbots, assistants.
- **High throughput** required for handling many users.
- **Batching** multiple requests helps, but requires smart scheduling.

What Are the Key Challenges in Serving Large Language Models?

3. Dynamic Workloads

- **Unpredictable request patterns:** Some prompts are short, others long.
- **Output length varies per request,** making resource allocation complex.

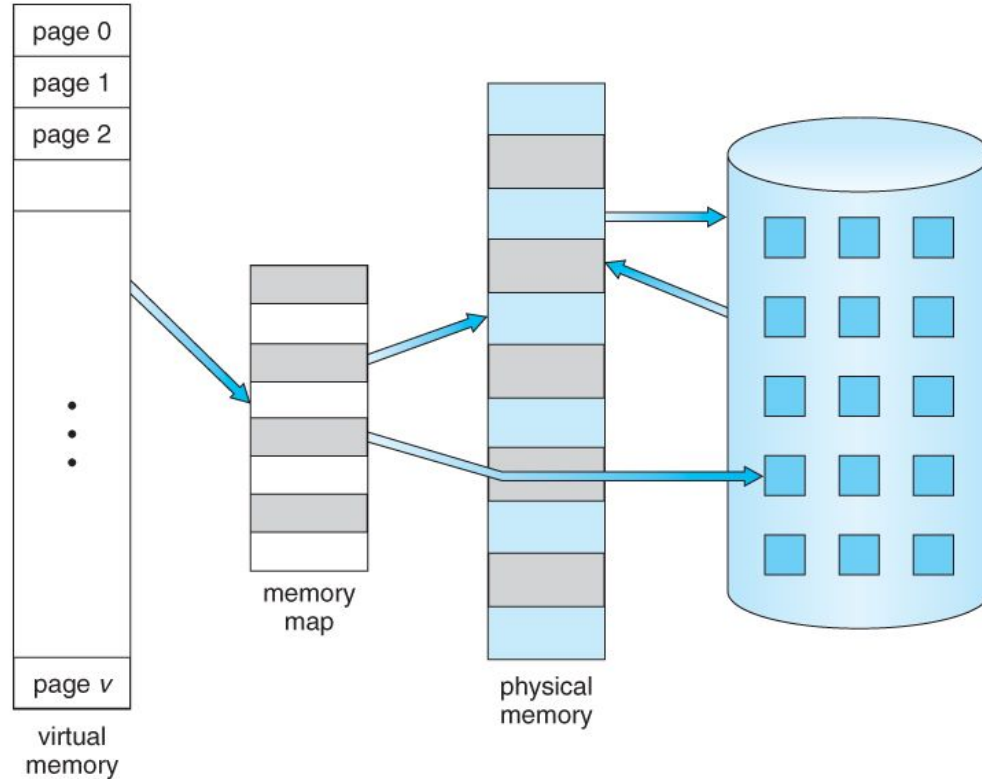
4. Hardware Limitations

- **LLMs need expensive GPUs/TPUs.**
- **Scaling across multiple GPUs introduces communication overhead.**
- **Autoscaling is needed to manage fluctuating workloads efficiently.**

OS Primer: Virtual Memory and Paging

Inspiration to vLLM

OS Primer: Virtual Memory and Paging



What is Virtual Memory?

- **Virtual Memory** is an OS mechanism that gives applications the illusion of having more memory than physically available.
- It allows efficient **memory management** by **mapping virtual addresses to physical memory pages**.

How It Works:

1. **Processes get virtual addresses** instead of direct access to physical memory.
2. **Memory pages are allocated dynamically** as needed.
3. **Unused pages can be swapped** to disk (swap space) when memory is full.
4. **Page tables track the mapping** between virtual and physical addresses.

Key Benefits:

- ✓ **Efficient memory use** – No need to allocate all memory upfront.
- ✓ **Memory isolation** – Prevents processes from interfering with each other.
- ✓ **Supports large applications** – Programs can use more memory than physically available.

What is Paging?

- **Paging** is a memory management technique that breaks memory into **fixed-size blocks** called **pages** (in virtual memory) and **page frames** (in physical memory).
- The OS loads only **needed pages** into memory instead of the entire program.

Paging Process:

1. **The program requests data** stored in virtual memory.
2. **The OS finds the page** in physical memory (RAM). If not found, a **page fault** occurs.
3. **If needed, pages are swapped** in and out between RAM and disk.

Paging vs. Contiguous Memory Allocation:

Feature	Paging	Contiguous Memory Allocation
Memory Efficiency	High (only needed pages are loaded)	Low (must allocate a fixed large block)
Fragmentation	Low (non-contiguous pages can be used)	High (requires contiguous space)
Flexibility	Can dynamically allocate/release memory pages	Hard to resize once allocated

How Does Virtual Memory Improve Performance?

Problems with Traditional Memory Allocation:

- **Internal Fragmentation:** Unused space within allocated memory.
- **External Fragmentation:** Gaps of free memory that are too small to be used.
- **Fixed Allocations:** Programs must pre-allocate memory, leading to waste.

How Virtual Memory Helps:

- ✓ **Paging eliminates fragmentation** by using fixed-size blocks.
- ✓ **Allows multi-tasking** by efficiently sharing memory across applications.
- ✓ **Improves scalability** by offloading unused memory pages to disk.

Inspiration for vLLM – Bringing OS Paging to LLM Inference

Traditional LLM Serving Issues:

- **LLMs allocate large KV cache memory** in contiguous blocks.
- **Memory fragmentation leads to inefficiency** (some allocated memory is wasted).
- **Fixed allocation per request limits batching**, reducing throughput.

How vLLM Uses Paging Concepts:

- **PagedAttention works like virtual memory** – It breaks KV cache into **small blocks ("pages")** instead of one large contiguous chunk.
- **Dynamically allocates memory pages** for active requests only, preventing waste.
- **Frees memory pages when no longer needed**, enabling efficient multi-request batching.
- **Inspired by OS paging, but optimized for GPU memory** rather than CPU RAM.

The vLLM Paper

Efficient Memory Management for Large Language
Model Serving with PagedAttention

Abstract Overview

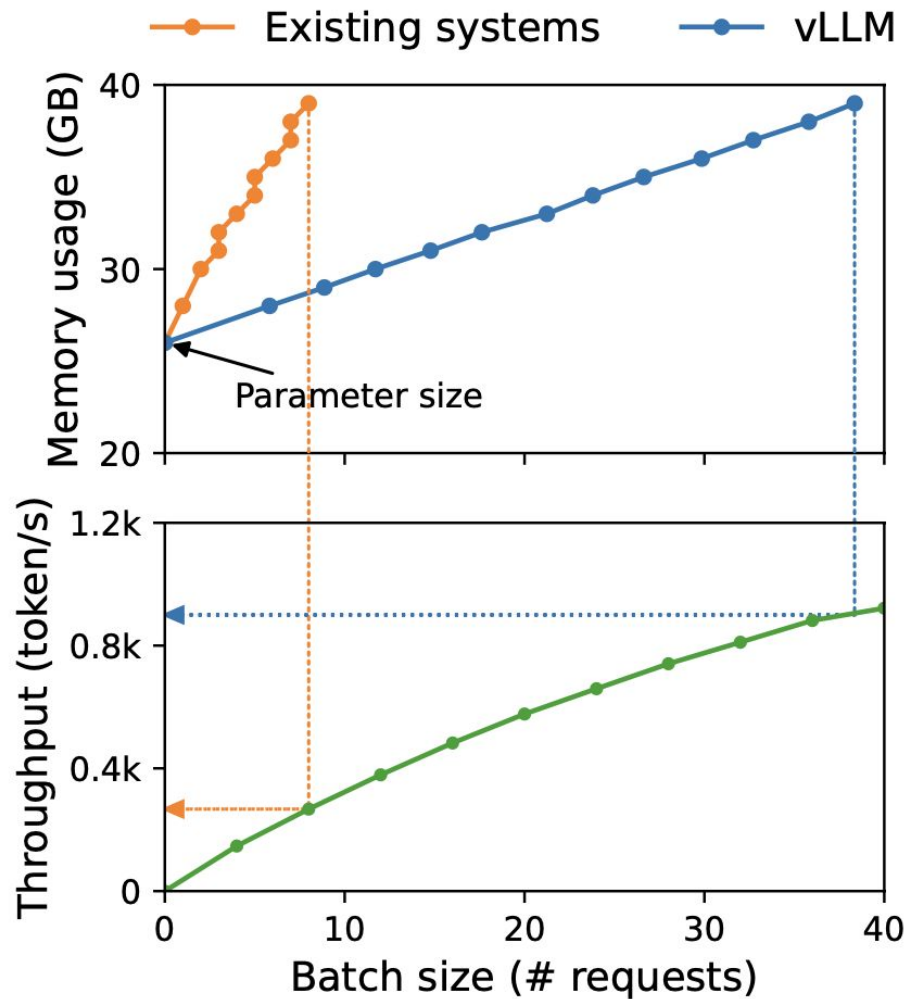
Problem: Efficiently batching LLM requests is challenging due to large and dynamically changing key-value (KV) cache memory.

Solution: vLLM introduces **PagedAttention**, an attention mechanism inspired by virtual memory paging, optimizing memory usage.

Key Outcomes:

- Near-zero waste in KV cache memory.
- Flexible KV cache sharing within and across requests.
- 2-4× throughput improvement over FasterTransformer and Orca.

Abstract Overview



Challenges in LLM Serving

Autoregressive Transformers:

- LLMs generate tokens sequentially, creating a memory-bound workload.
- GPUs remain underutilized despite high computation power.

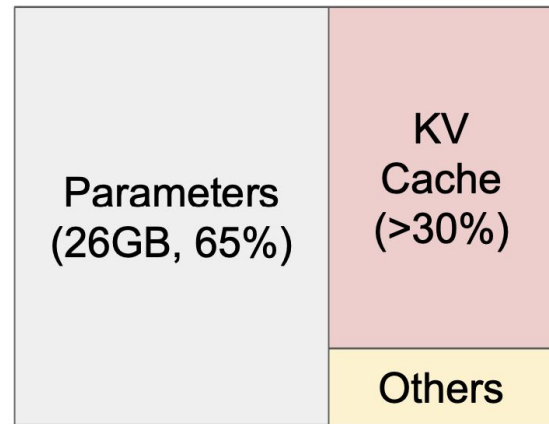
Batch Processing Solution:

- Batching multiple requests increases efficiency.
- However, **memory allocation inefficiencies** limit batch sizes.

GPU Memory Usage Breakdown

Memory Allocation for a 13B-parameter LLM on an NVIDIA A100 (40GB RAM):

- **65%** - Model Weights (Static, unchanged during inference)
- **30%** - Dynamic KV Cache (Attention memory)
- **5%** - Other data (activations, intermediate tensors)



NVIDIA A100 40GB

Key Insight:

- The **KV cache is the bottleneck** for increasing batch size and throughput.

GPU Memory Usage Breakdown

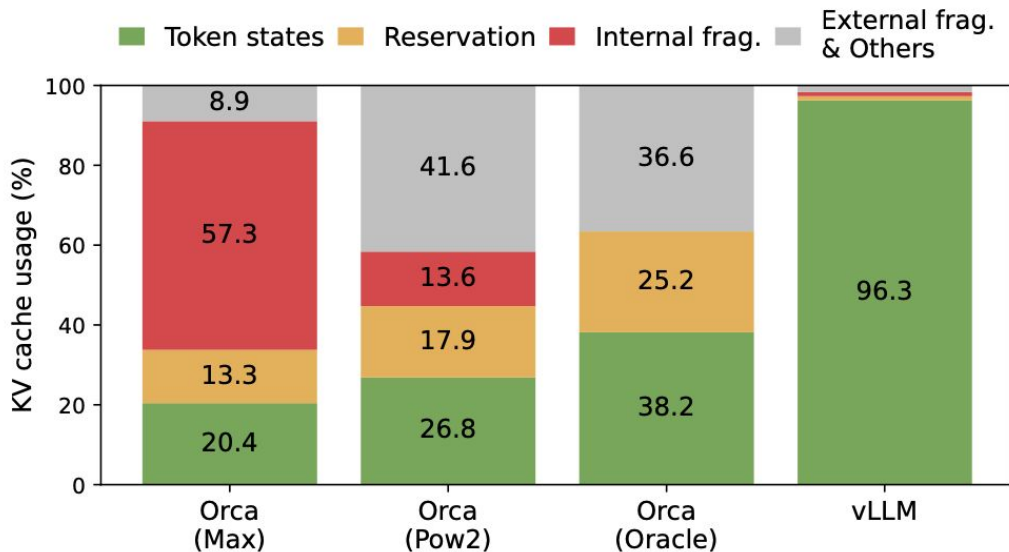


Figure 2. Average percentage of memory wastes in different LLM serving systems during the experiment in §6.2.

KV Cache Management Issues

Why Current Systems Fail:

1. **Memory Fragmentation:**

- Pre-allocating contiguous memory chunks leads to wasted space.
- Internal & external fragmentation result in only **20.4%-38.2% memory utilization**.

2. **Lack of Memory Sharing:**

- Advanced decoding methods (parallel sampling, beam search) generate multiple outputs per request.
- Existing systems do not share KV cache efficiently across these sequences.

KV Cache Management Issues

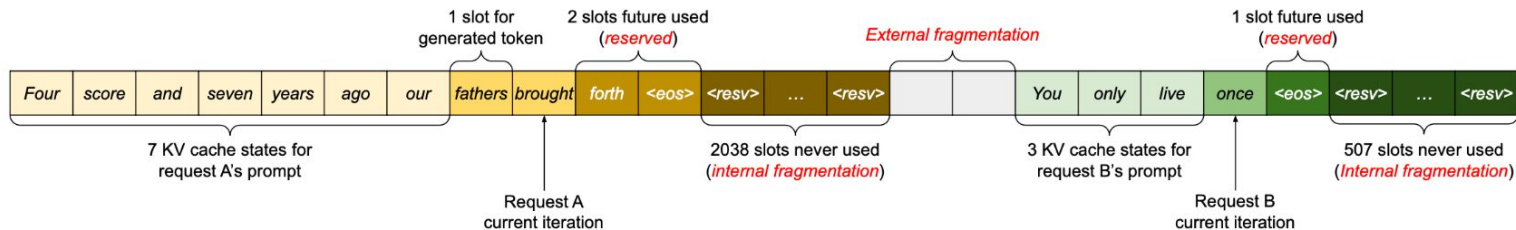


Figure 3. KV cache memory management in existing systems. Three types of memory wastes – reserved, internal fragmentation, and external fragmentation – exist that prevent other requests from fitting into the memory. The token in each memory slot represents its KV cache. Note the same tokens can have different KV cache when at different positions.

Introducing PagedAttention

Inspired by Virtual Memory Paging:

- **PagedAttention** stores KV cache in small, non-contiguous memory blocks.
- Treats **blocks as pages, tokens as bytes, requests as processes**.
- Enables **efficient dynamic allocation** and eliminates fragmentation.

Benefits:

- ✓ Reduced internal fragmentation
- ✓ No external fragmentation
- ✓ Memory sharing across sequences & requests

Introducing PagedAttention

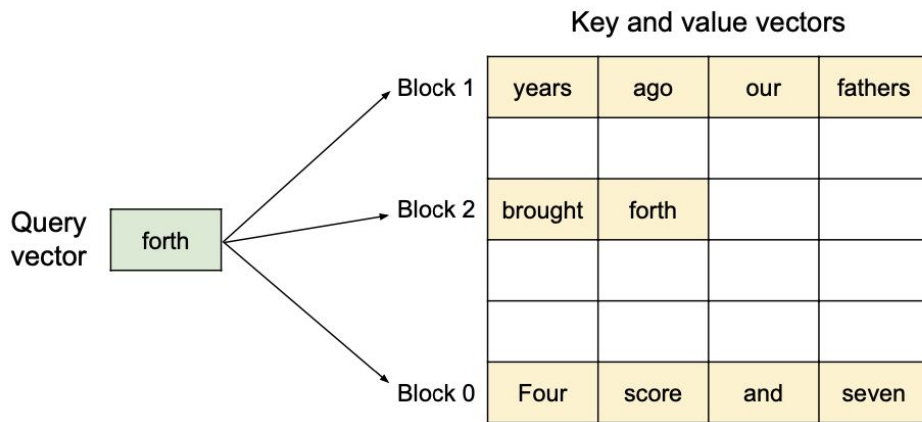


Figure 5. Illustration of the PagedAttention algorithm, where the attention key and values vectors are stored as non-contiguous blocks in the memory.

vLLM - High-Throughput LLM Serving Engine

Built on PagedAttention, vLLM achieves:


- **Near-zero KV cache waste** via block-level memory management.
- **Preemptive request scheduling** for efficient execution.
- **Support for popular models** (GPT, OPT, LLaMA) at any scale, including multi-GPU setups.

Performance Gains

Evaluation Results:

- **2-4× throughput improvement** over FasterTransformer and Orca.
- No loss in model accuracy.
- Gains increase for **longer sequences, larger models, and complex decoding algorithms.**

Open Source Availability

- vLLM's source code is publicly available at:
 [**https://github.com/vllm-project/vllm**](https://github.com/vllm-project/vllm)

vllm-project/vllm

A high-throughput and memory-efficient inference
and serving engine for LLMs



 859

Contributors

 1k

Issues

 419

Discussions

 40k

Stars

 6k

Forks



LLM as a Conditional Generation Service

- Deployed as completion APIs or chatbots.
- Takes an input prompt and generates output tokens sequentially.
- The generated sequence consists of both prompt and output tokens.

Sequential Token Generation

- New tokens are generated one-by-one.
- Each new token depends on all previous tokens.
- **Key-Value (KV) Cache:**
 - a. Stores key and value vectors of existing tokens.
 - b. Used to generate future tokens efficiently.
 - c. The KV cache of a token varies based on its position in the sequence.

Phases of Generation Computation

1. Prompt Phase

- Processes the entire user prompt.
- Computes probability for the first new token.
- Generates key and value vectors for all prompt tokens.
- Efficient due to parallel matrix-matrix multiplication.
- Fully utilizes GPU parallelism.

Autoregressive Generation Phase

- Generates subsequent tokens sequentially.
- Each iteration processes one new token at a time.
- **Key Challenges:**
 - a. Cached KV vectors reduce redundant computation.
 - b. Data dependency prevents parallelization.
 - c. Relies on matrix-vector multiplication, leading to GPU underutilization.
 - d. Memory-bound operations contribute to request latency.

Phases of Generation Computation

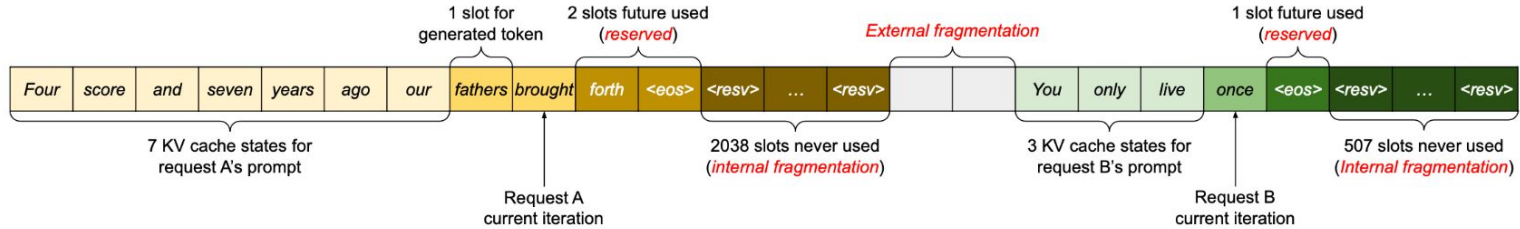


Figure 3. KV cache memory management in existing systems. Three types of memory wastes – reserved, internal fragmentation, and external fragmentation – exist that prevent other requests from fitting into the memory. The token in each memory slot represents its KV cache. Note the same tokens can have different KV cache when at different positions.

Batching Techniques for LLMs

- Multiple requests are grouped together in a batch.
- Shared model weights reduce computation overhead.
- Large batch sizes amortize weight movement costs.

Challenges in Batching

- **Asynchronous Request Arrival:**
 - a. Requests arrive at different times.
 - b. Simple batching strategies introduce queuing delays.
- **Variable Sequence Lengths:**
 - a. Different input/output lengths.
 - b. Padding wastes GPU computation and memory.

Fine-Grained Batching Mechanisms

Iteration-Level Scheduling

- Unlike request-level batching, operates at the iteration level.
- After each iteration:
 - Completed requests are removed from the batch.
 - New requests are added immediately.
- Reduces queuing delays significantly.
- Optimized GPU kernels eliminate padding inefficiencies.
- Leads to higher throughput in LLM serving.

Compaction Limitations

- Proposed as a solution for memory fragmentation.
- **Challenges in Performance-Sensitive Systems:**
 - a. Large KV cache prevents efficient compaction.
 - b. Pre-allocated chunk spaces hinder memory sharing.
 - c. Not practical for real-time LLM serving systems.

vLLM System Architecture

- **Key components:**
 - **Centralized Scheduler:** Manages execution across GPUs.
 - **KV Cache Manager:** Handles KV cache in a paged manner.
 - **PagedAttention:** Enables non-contiguous KV cache allocation.

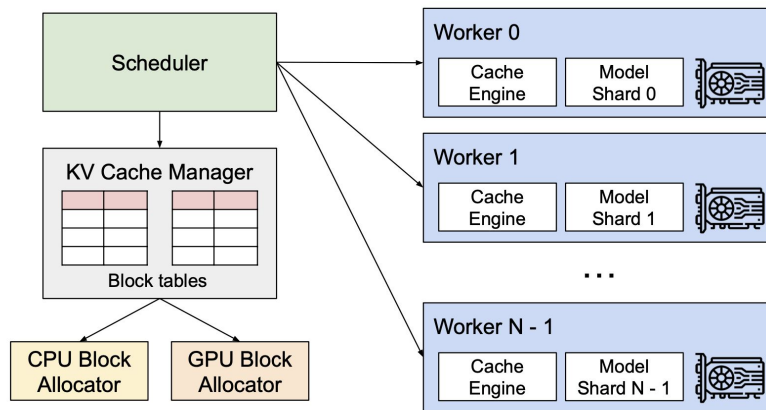


Figure 4. vLLM system overview.

PagedAttention - The Core Algorithm

- **Inspired by OS paging mechanisms**
- Traditional attention vs. PagedAttention:
 - a. Traditional: Stores KV cache in contiguous memory.
 - b. **PagedAttention**: Divides KV cache into **KV blocks**, stored non-contiguously.

How PagedAttention Works

- **Step-by-step execution:**
 - a. Kernel identifies required **KV blocks**.
 - b. **Computes attention** per block.
 - c. Multiplies **query vector** with **key vectors** of a block.
 - d. Multiplies **attention scores** with corresponding **value vectors**.
- **Key Benefit:** Memory efficiency through flexible, paged memory allocation.

KV Cache Manager - Virtual Memory for LLMs

- **Analogy to OS Virtual Memory:**
 - a. Logical pages (KV blocks) → Non-contiguous physical memory.
 - b. No need to reserve full KV cache in advance.
- **Memory allocation strategy:**
 - a. KV cache grows dynamically.
 - b. Blocks are **allocated and mapped on demand**.
 - c. **Reduces memory waste** compared to traditional allocation.

GPU and CPU KV Cache Allocation

- **GPU Workers:**

- Allocate a **contiguous** GPU DRAM chunk.
- Divide it into **physical KV blocks**.

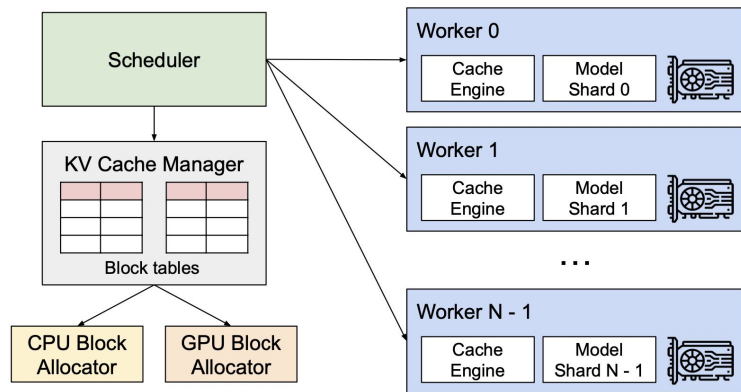


Figure 4. vLLM system overview.

- **CPU RAM Swapping:**

- Enables **efficient swapping** for long sequences.
- Block tables map **logical KV blocks** → **physical KV blocks**.

Scheduling and Preemption

First come first serve scheduling to provide fairness among requests and non-starvation.

Challenges

- Input and output lengths are not known a priori, so vLLM can run out of GPU physical blocks for new KV Cache
 - 1 - Which blocks should it evict?
 - How to recover evicted blocks if needed?

Preemption Strategy

All-or-Nothing Eviction

- Either evict all blocks of a sequence or none
- Latest arrived requests are preempted first

Gang Scheduling

- Multiple sequences within one reqs. are treated as **sequence group** and always preempted/scheduled together

After preemption, vLLM stops accepting new reqs. until preempted seq. are complete

When a block completes, its blocks are freed

Recovery Techniques

Swapping

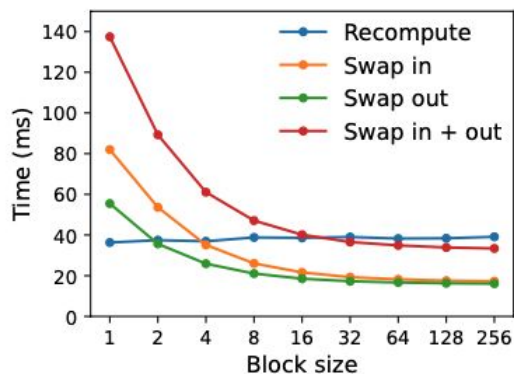
- Evicted blocks are copied to CPU RAM
- Managed by a dedicated CPU block allocator
- Swap space is bounded by GPU memory section of KV Cache

Recomputation

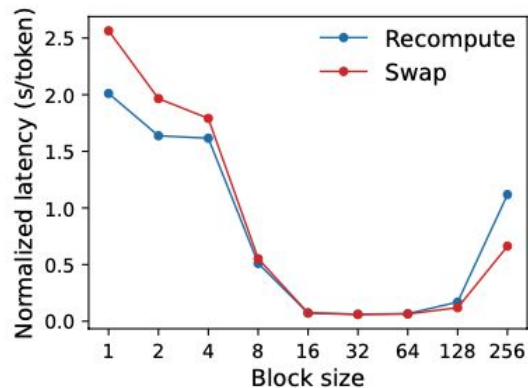
- KV cache is recomputed when preempted seq. is rescheduled
- Can be faster, interestingly

Comparing Recomputation and Swapping

- Small block sizes increase data transfer, limiting the effective bandwidth
- Recomputation is efficient when block size is small
- As block is getting bigger, swap makes sense



(a) Microbenchmark

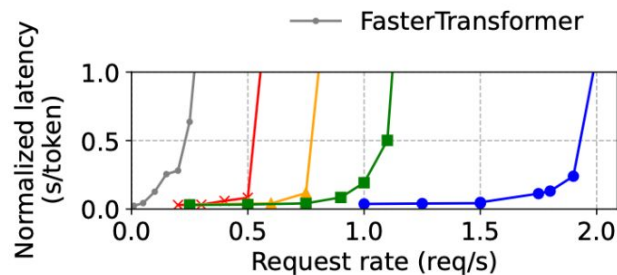


(b) End-to-end performance

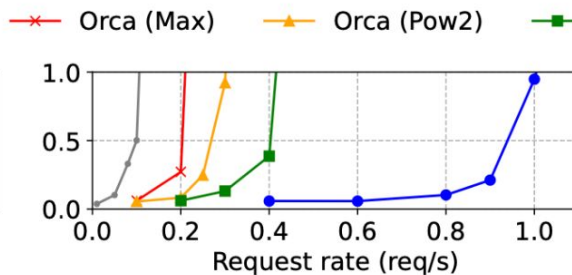
Distributed Inference

- When do we need it?
- Tensor parallelism(GPU per node), pipeline parallelism(node count)
- Single KV cache manager within central scheduler
- How does it work?
 1. Scheduler prepares message with input token IDs and block tables
 2. Control message is broadcast to all GPU workers
 3. GPU workers execute the model with input
 4. Workers use single KV cache
 5. Workers synchronize intermediate results using all-reduce

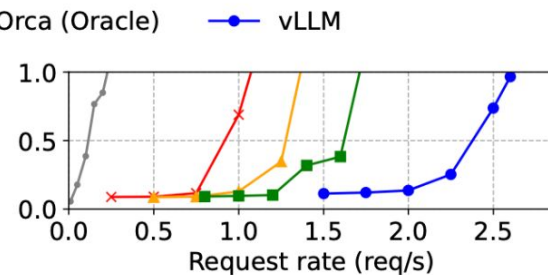
Evaluation



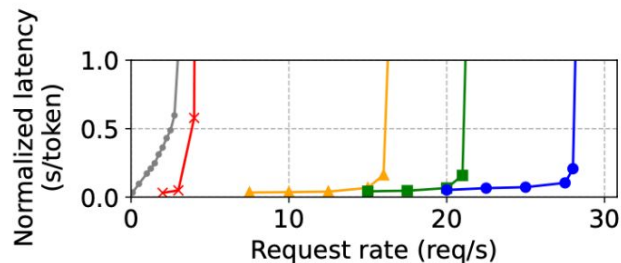
(a) OPT-13B, 1 GPU, ShareGPT



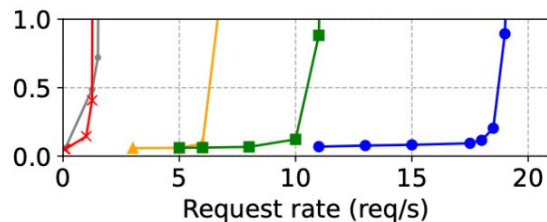
(b) OPT-66B, 4 GPUs, ShareGPT



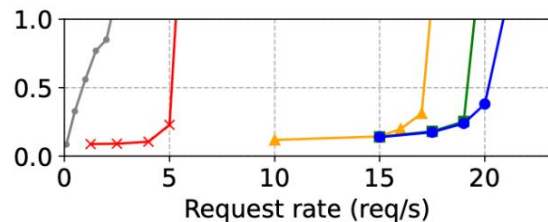
(c) OPT-175B, 8 GPUs, ShareGPT



(d) OPT-13B, 1 GPU, Alpaca

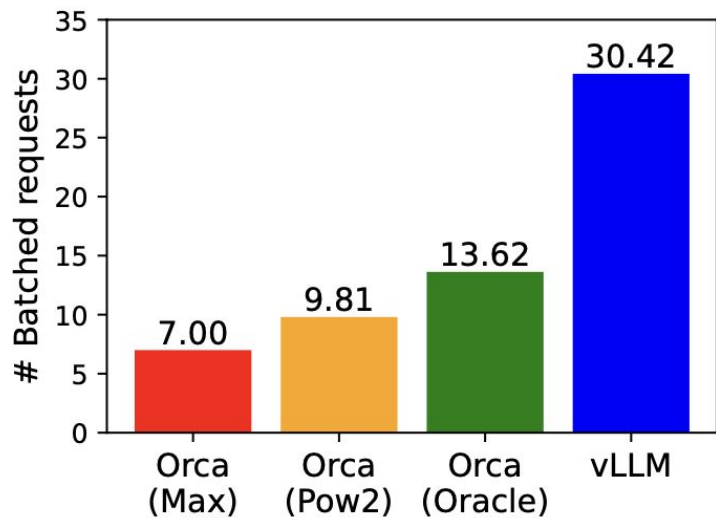


(e) OPT-66B, 4 GPUs, Alpaca

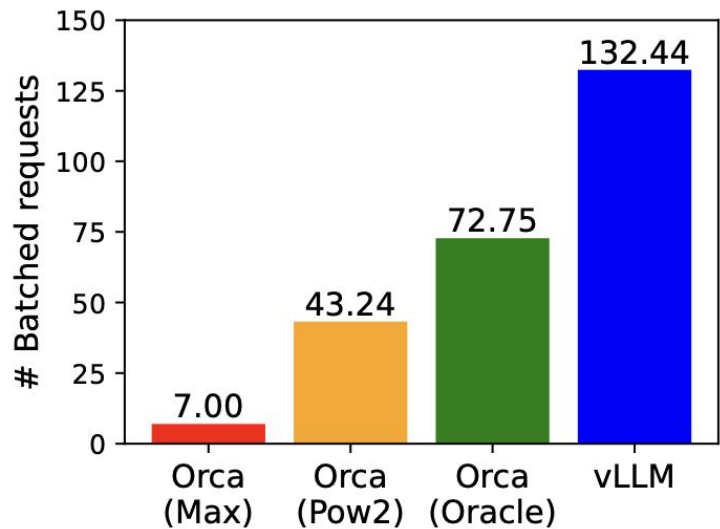


(f) OPT-175B, 8 GPUs, Alpaca

Evaluation



(a) ShareGPT

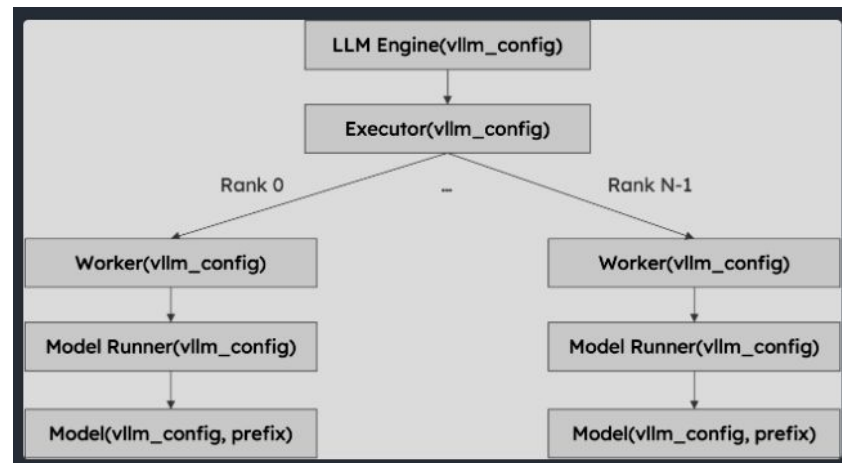
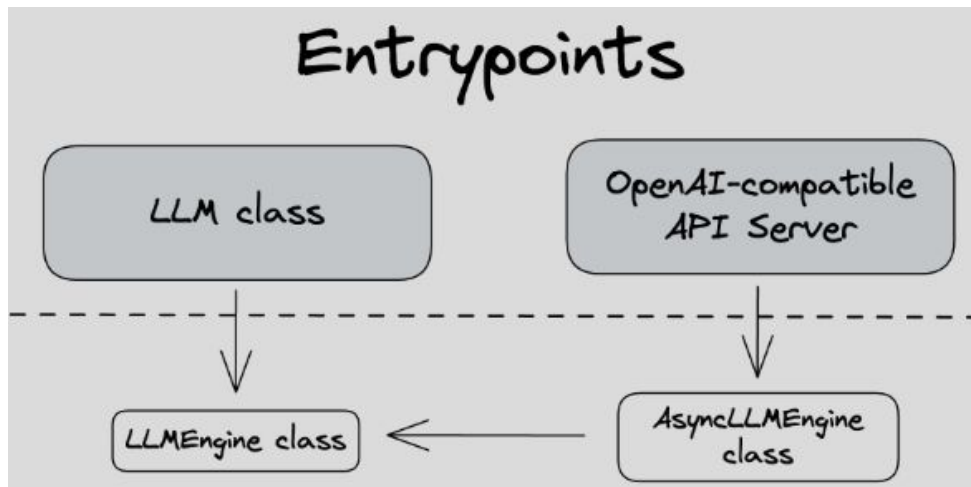


(b) Alpaca

vLLM Architecture

What are the key components?

Entrypoints



Entrypoints

LLM Class

- Python interface for offline inference
- Ideal for batch processing and app integration

```
from vllm import LLM, SamplingParams  
llm = LLM(model="facebook/opt-125m")  
outputs = llm.generate(prompts, sampling_params)
```

OpenAI Compatible API Server

- Rest API server comp. with OpenAI API Spec
- Makes it easier to integrate with existing apps

```
vllm serve <model>
```

Core Engine

This layer sits between user facing entrypoints and exec. infra coordinating inference process

LLM Engine

- input processing: tokenizes input text using model tokenizer
- scheduling: determines which requests are processed in each step
- model execution: manages execution, including distributed ops.
- output processing: decodes model outputs into human readable f.

AsyncLLMEngine

- Async wrapper around LLMEngine for online serving
- Uses `asyncio` for continuous processing of concurrent reqs.
- Enables streaming outputs to clients
- Used by OpenAI compatible API Server

Execution Infrastructure

Where the real magic happens

Worker Layer

- One worker process controls one accelerator(GPU etc.), enables parallel execution across multiple GPUs
- Scale with parallelism (tensor, pipeline)
- Identified by rank(global orchestration) and local rank(assigning device, accessing local resource)

Model Runner

- Bridging worker and model, provides consistent interface
- Loads model weights and configs
- Prepares input tensors and captures CUDA graphs
- Manages memory and computational resources

Model

- The actual PyTorch module

Deployment & Benchmarking