vLLM

Efficient Memory Management for Large Language Model Serving with PagedAttention

Chenye Wang

Sept 22, 2024

[1] Efficient Memory Management for Large Language Model Serving with PagedAttention. SOSP, 2023.

Why do we need vLLM?

The Era of LLMs

- LLM-Powered Services are increasingly important
- However...
  - very expensive
  - a large number of GPUs
- Solution
  - high **throughput** to reduce cost

ChatGPT

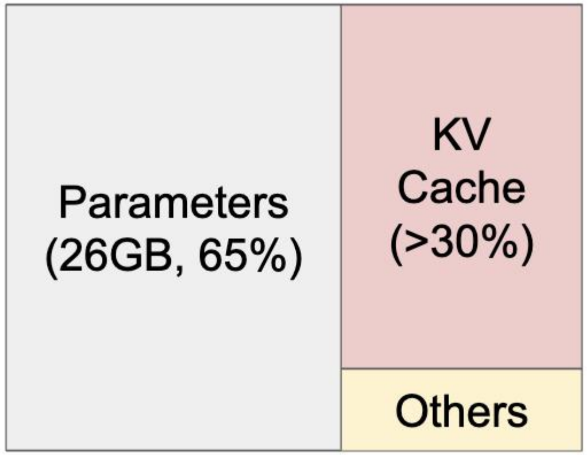
GitHub Copilot

Challenge

- **Batching requests** can improve throughput
  - memory-bound
  - need efficient management for requests' memory -- KV cache
    - huge
    - dynamic
    - unknown lifetime and length

- How do the existing LLM serving systems do?
  - store the KV cache in *contiguous* memory space
  - *pre-allocate* chunk

- What are the problems?

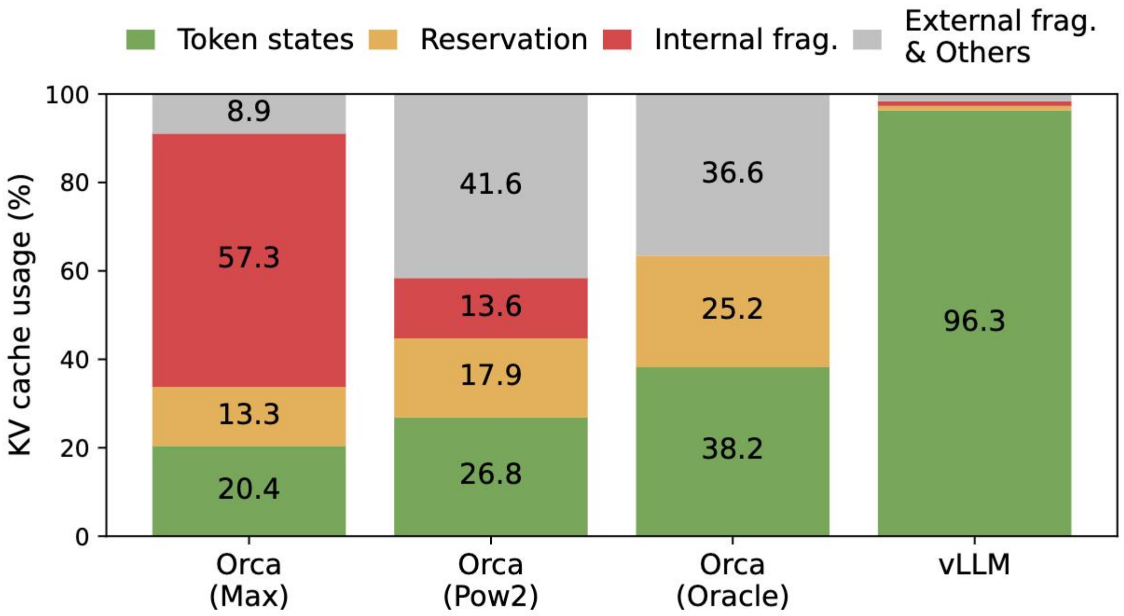


NVIDIA A100 40GB

Memory Fragmentation

- Internal fragmentation
  - actual length can be much shorter
  - inefficiency
    - other requests cannot utilize the part that is currently unused

- External fragmentation
  - different size
  - e.g., buddy allocator



Memory Sharing

- Advanced LLM decoding algorithms generate multiple outputs for one prompt
    - thus sequences can partially share their KV cache
    - e.g., parallel sampling, beam search
- But...
    - KV cache is stored in separate contiguous spaces

How does vLLM do?

PagedAttention

- Inspired by OS
    - ***virtual memory and paging techniques***
- Divide KV cache into **blocks**
    - contain a fixed number of tokens
    - not necessarily stored in contiguous space
- Analogy
    - page to block
    - byte to token
    - process to request

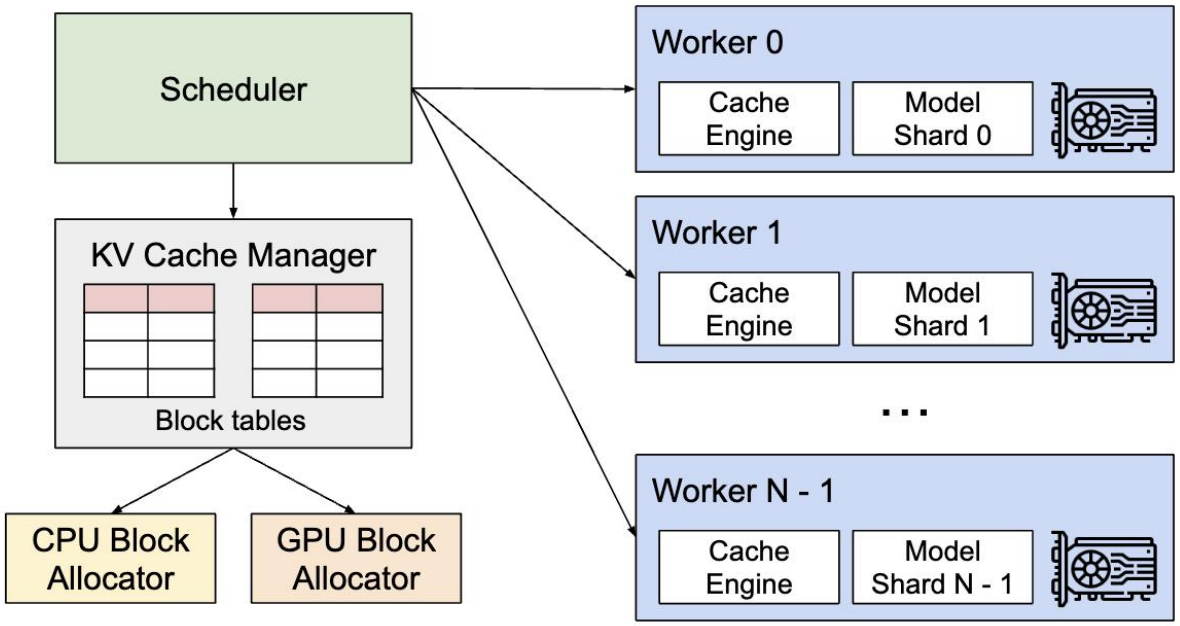
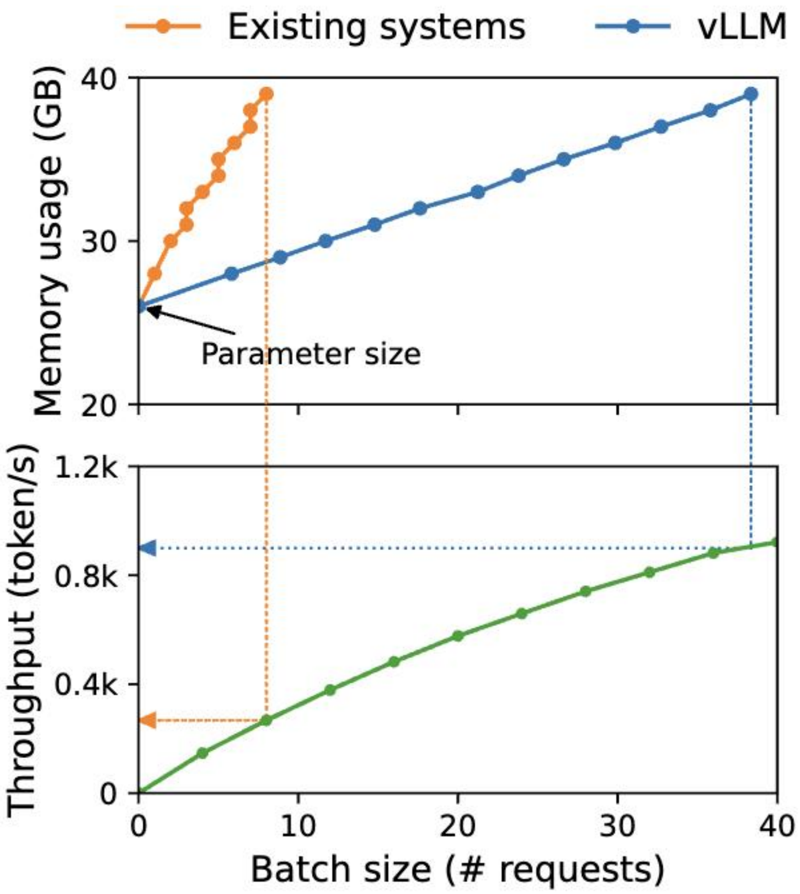


Figure 4. vLLM system overview.

Problems before?

- Internal fragmentation?
  - use small blocks
  - allocate on demand
- External fragmentation?
  - same block size
- memory sharing?
  - at the granularity of a block
  - block mapping



Memory with PagedAttention

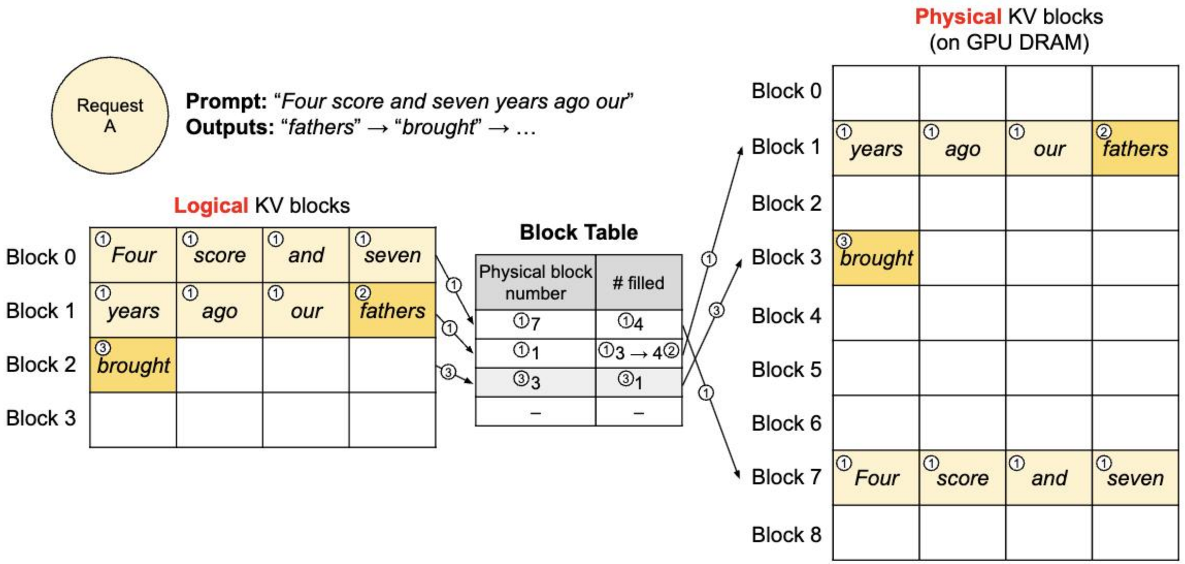


Figure 6. Block table translation in vLLM.

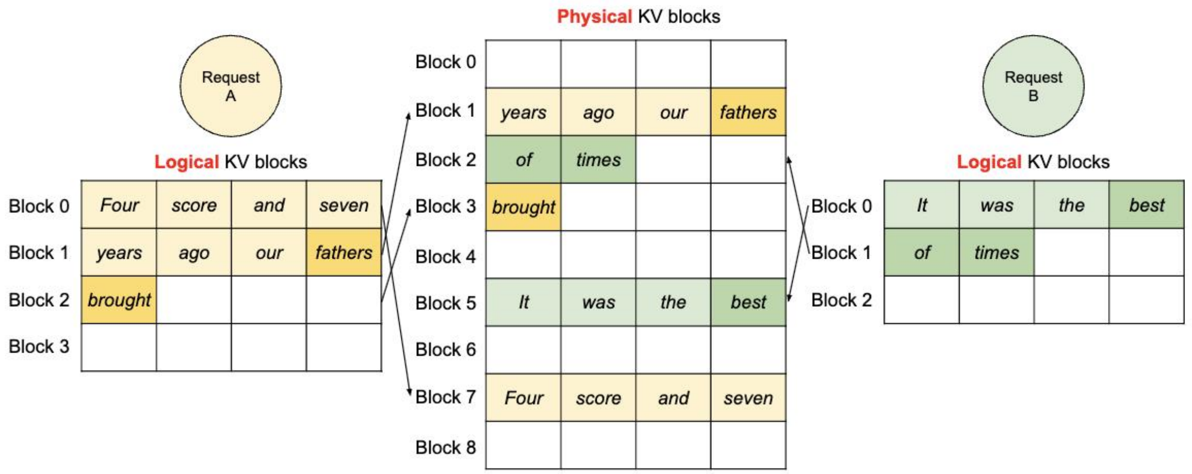


Figure 7. Storing the KV cache of two requests at the same time in vLLM.

Copy-on-Write

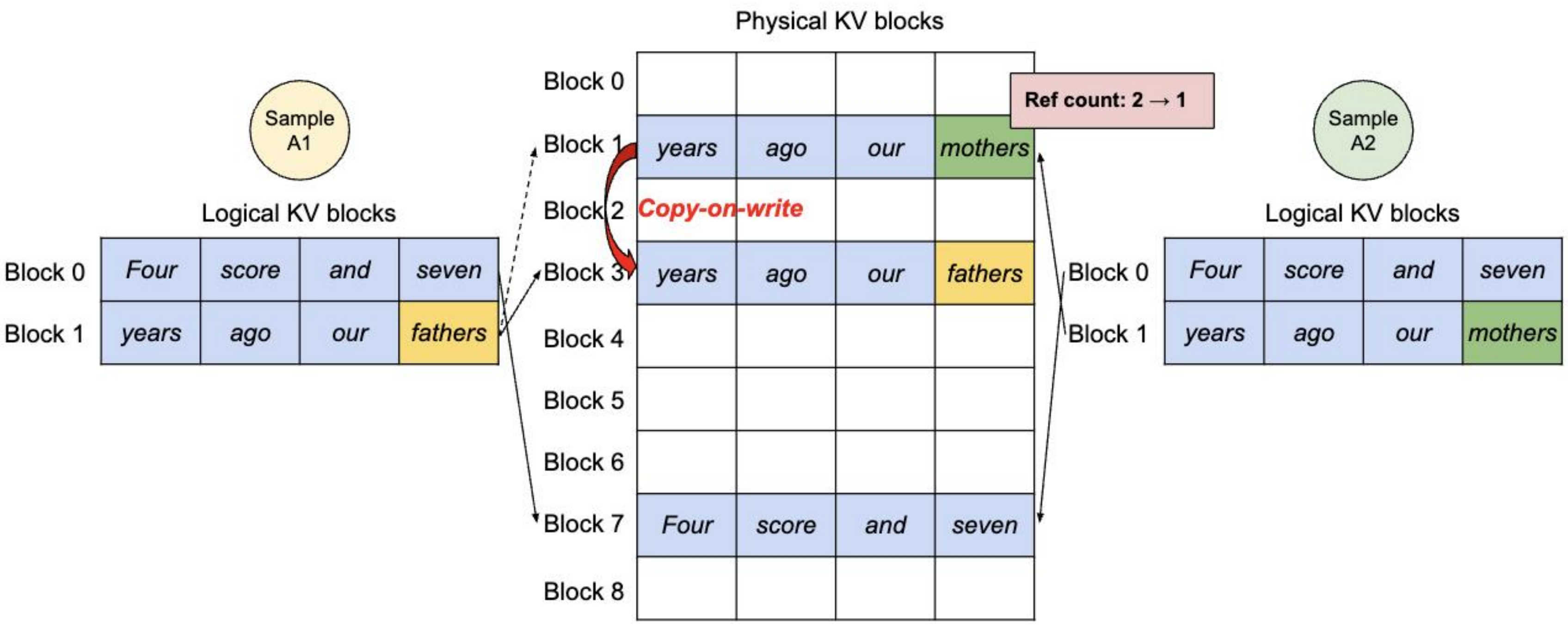


Figure 8. Parallel sampling example.

Other applications

- Beam search
  - similar to *process tree*

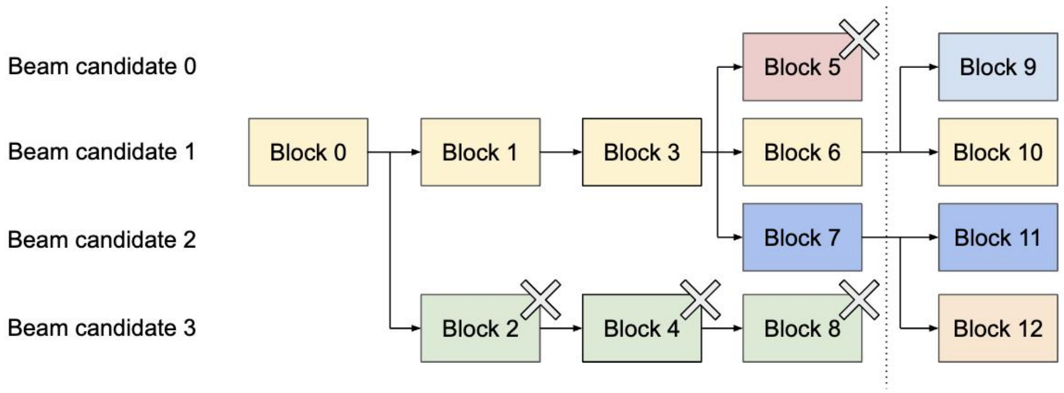


Figure 9. Beam search example.

- Shared prefix
  - similar to *shared library*

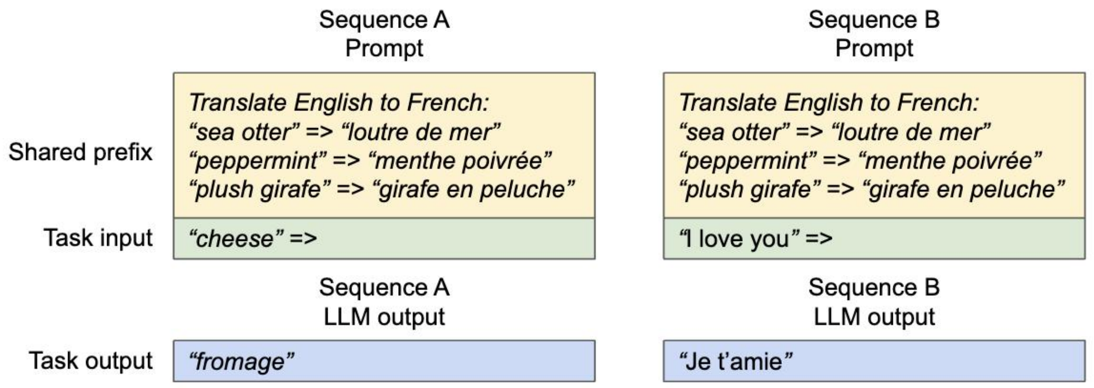


Figure 10. Shared prompt example for machine translation. The examples are adopted from [5].

Kernel-level Optimization

- **Indirect memory access pattern** of PagedAttention is not efficiently supported by existing systems
  - develop custom kernels to optimize it

- fuse operations into kernels
  - KV cache reshape and block write
  - block read and attention operations
  - block copy

Support Various Decoding Algorithms

- vLLM implements various decoding algorithms using three key methods:
  - *fork*
    - create a new sequence from an existing one
  - *append*
    - append a new token to the sequence
  - *free*
    - delete the sequence

- Think of process in OS

What if requests exhaust GPU memory?

Scheduling and Preemption

- Scheduling
    - FCFS
        - ensure fairness
        - prevent starvation

- Preemption
    - the earliest arrived requests are served first
    - the latest requests are preempted first

Which blocks should vLLM evict?

- the block will be accessed furthest in the future

- **all-or-nothing** eviction policy
  - either evict all or none of the blocks of a sequence
  - multiple sequences within one request are preempted or rescheduled together as a sequence group

How to recover evicted blocks?

- Swapping
  - select a set of sequences to evict, copy their blocks to CPU memory
  - stop accepting new requests until all preempted sequences are completed
  - free blocks of completed requests
  - bring blocks of preempted sequences back
- Recomputation
  - concatenate tokens generated before with original prompt
  - recompute the KV cache

Thinking and Discussion

Tradeoff

- Kernel
  - higher attention kernel latency
    - access the block table
    - execute extra branches
- Block Size
  - too small
    - not fully utilize GPU’s parallelism
  - too large
    - internal fragmentation
    - sharing chances decrease
- Recovery mechanisms
  - Swapping
    - depend on the bandwidth between CPU RAM and GPU memory
    - excessive overhead with small block sizes
  - Recomputation
    - depend on computation power of GPU

Discussion for virtual memory and paging

- Applying the same technique to other GPU workloads?
  - stored data shape is static?
  - performance is compute-bound?

- LLM-specific optimizations in applying virtual memory and paging
  - all-or-nothing swap-out policy
  - recomputation for KV cache
  - fusing the GPU kernels for memory access operation

Summary

- Strength
  - flexible memory management
  - near-zero memory waste
  - high hardware utilization and low latency
  - simultaneous processing with different decoding algorithms
- Weakness
  - cannot fundamentally improve inference latency
  - not support sharding the KV cache
  - memory capacity due to large model weights is still the bottleneck

Thanks!

Q & A