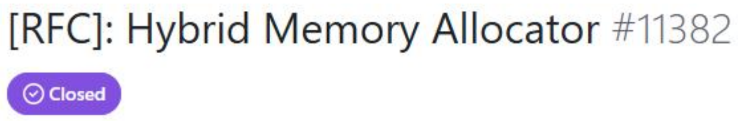
西北工業大學
NORTHWESTERN POLYTECHNICAL UNIVERSITY


[RFC]: Hybrid Memory Allocator #11382
Closed

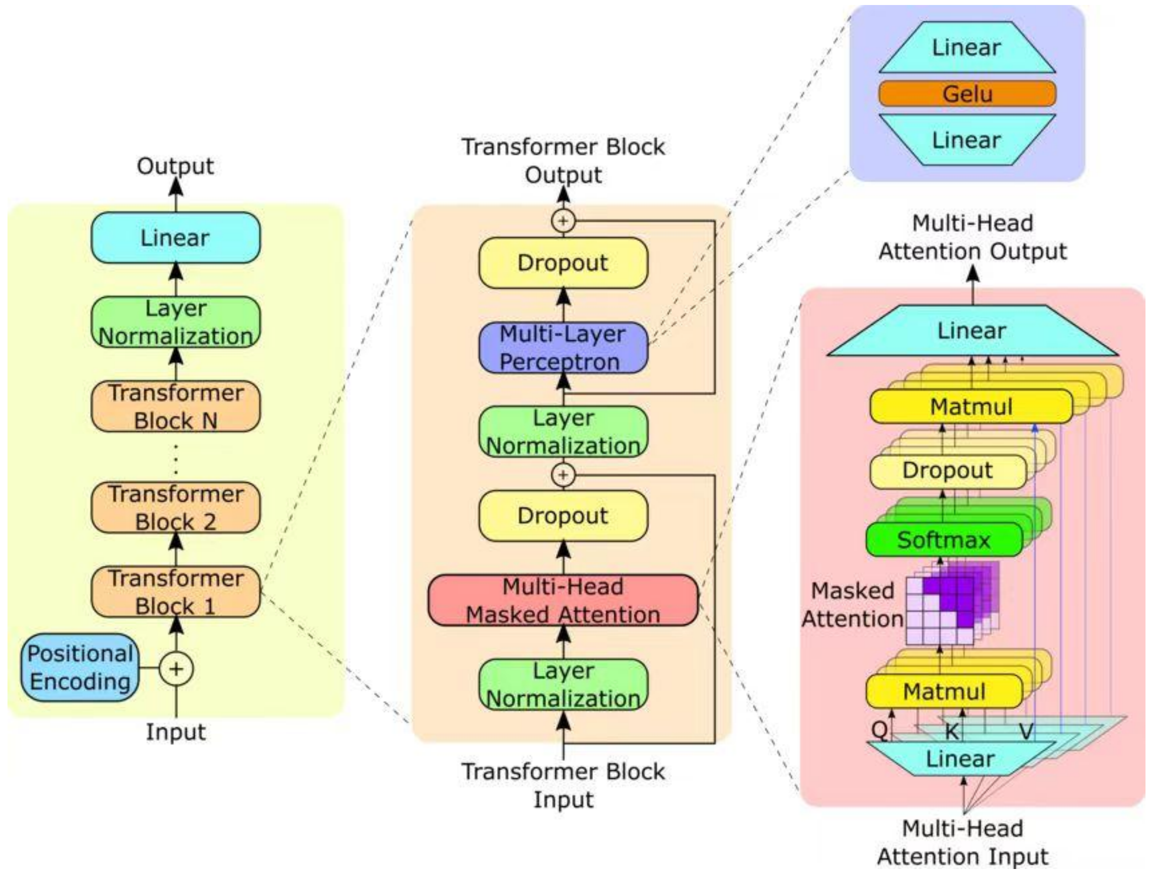
https://github.com/vllm-project/vllm/issues/11382.

Jenga: Effective Memory Management for Serving LLM with Heterogeneity

[1]*Tsinghua University* [2]*UC Berkeley* [3]*University of Chicago*
Presented by Mingxuan Liu, Northwestern Polytechnical University

January 6, 2026

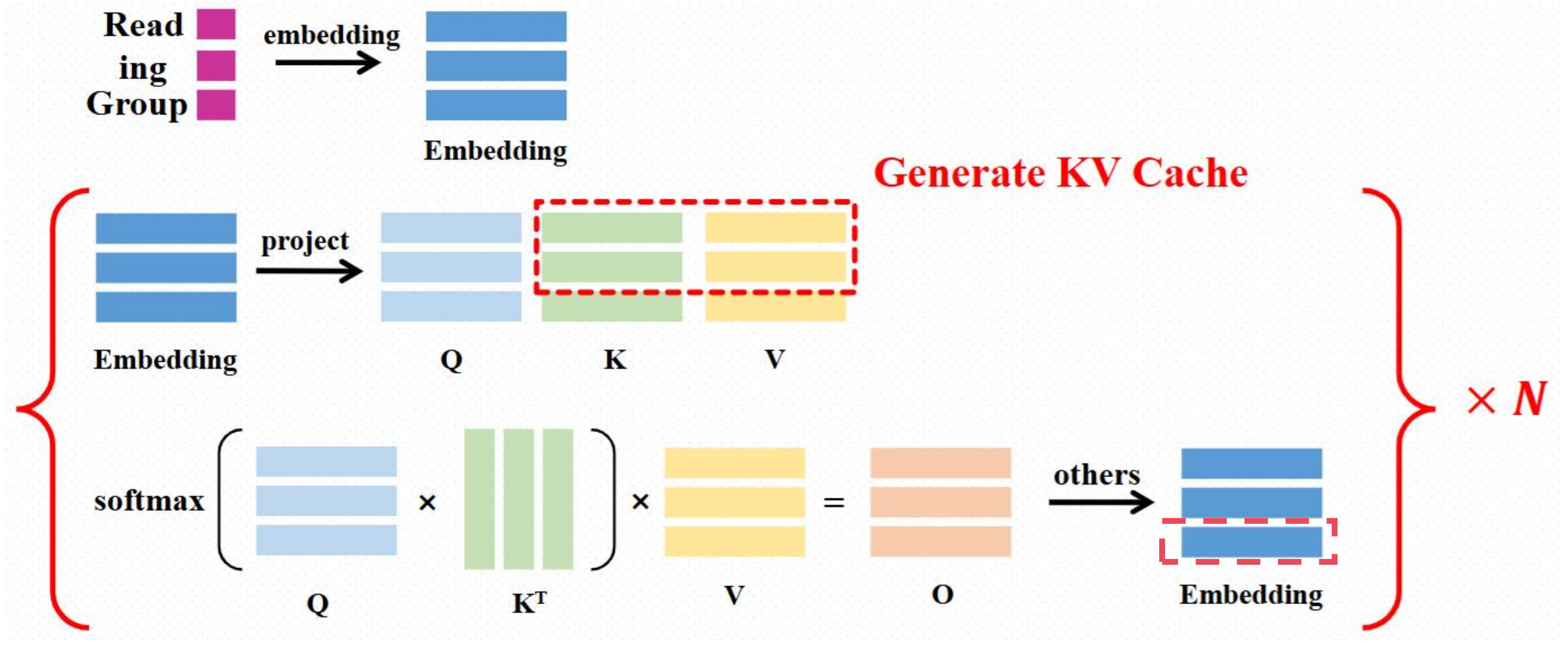
Background: Architecture of LLMs

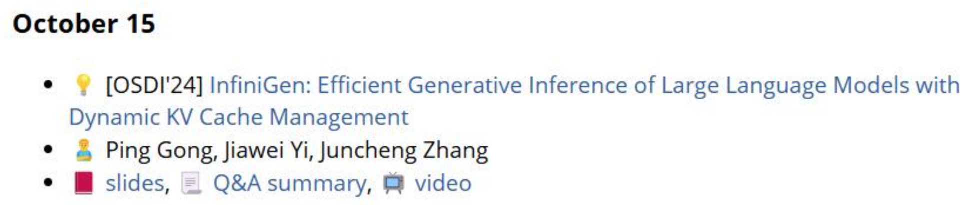


OPT-2 Architecture

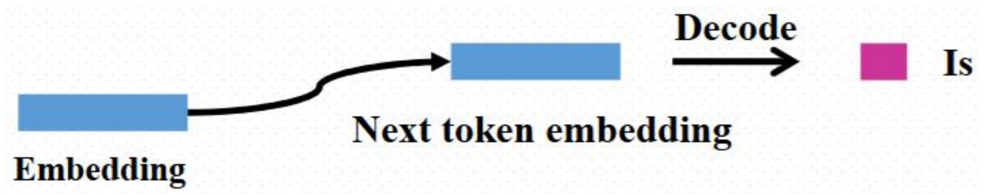
Background: Decoding in LLM Inference

- **Prefill: Generate KV cache & first token -> Compute-bound**



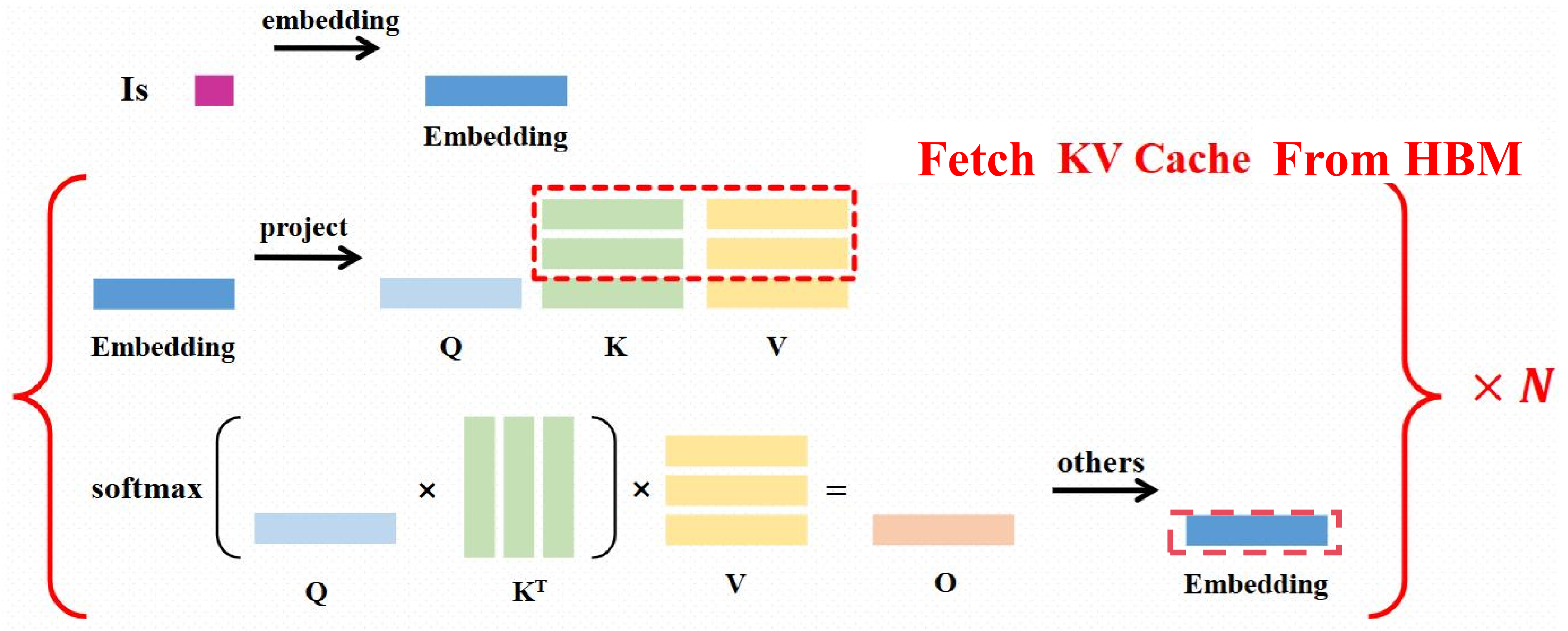

October 15

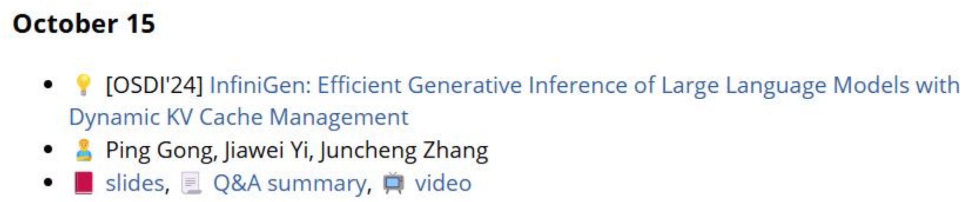
- 💡 [OSDI'24] InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management
- 👤 Ping Gong, Jiawei Yi, Juncheng Zhang
- 📕 slides, 📄 Q&A summary, 📺 video



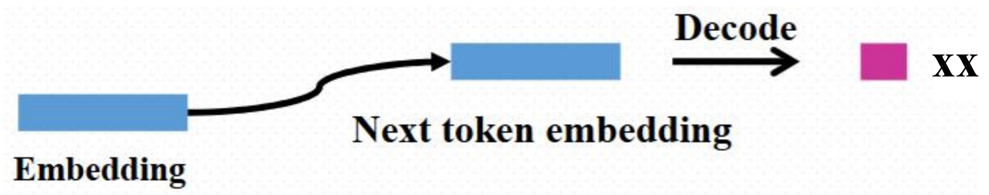
Background: Prefill in LLM Inference

- Decode: Fetch KV cache & generate next token



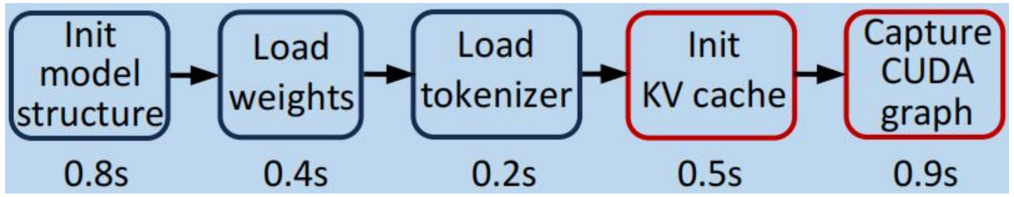

October 15

- 💡 [OSDI'24] InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management
- 👤 Ping Gong, Jiawei Yi, Juncheng Zhang
- 📕 slides, 📄 Q&A summary, 📺 video



Background: LLM Inference Engine

- **Startup[1]:**
  - S1: Init model structure (*.config*)
  - S2: Load weights
  - S3: Load tokenizer
  - **S4: Init KV Cache Memory** ★
    - Memory Profiling to calculate ***Available_KV_Cache_Memory*** and reserve
      - **Fixed-sized** during the lifestyle of LLM Engine (vLLM, SGLang, etc.)
  - S5: CUDA Graph Capturing

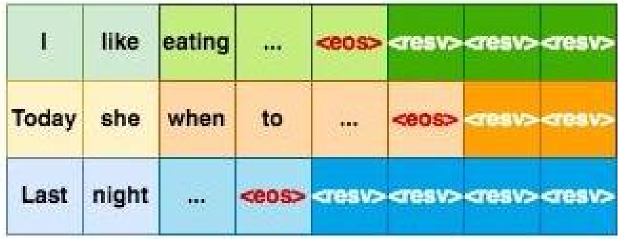


- **Inference: Scheduling + Computing**
  - Scheduling: Continues batching, Chunked Prefill, etc.
  - Computing: FlashAttention v3, etc.
- **Efficient *Available_KV_Cache_Memory* management** ★
  - maximize request batch size

1. Zeng, Shaoxun, et al. "Medusa: Accelerating serverless LLM inference with materialization." Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1. 2025.

Background: Paged Attention (1)

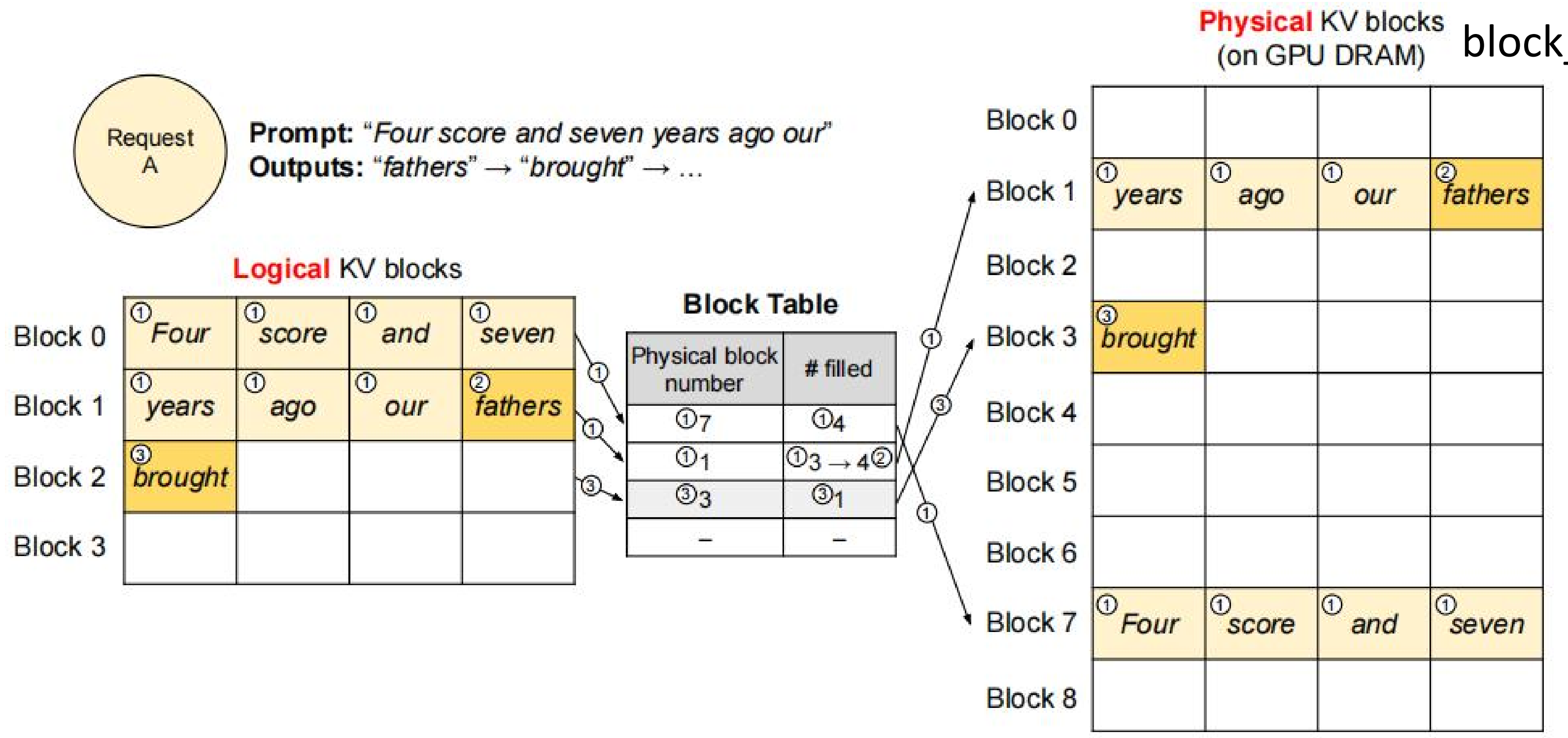
Traditional KVCache Alloc[1] for 3 Requests



prompt1	I	like	eating	...	<eos>	<resv>	<resv>	<resv>
prompt2	Today	she	when	to	...	<eos>	<resv>	<resv>
prompt3	Last	night	...	<eos>	<resv>	<resv>	<resv>	<resv>

PagedAttention KVCache Alloc[2] for 1 Request

block_size = 4



Request A
Prompt: "Four score and seven years ago our"
Outputs: "fathers" → "brought" → …
Logical KV blocks
Block 0: ① Four, ① score, ① and, ① seven
Block 1: ① years, ① ago, ① our, ② fathers
Block 2: ③ brought
Block 3
Physical KV blocks (on GPU DRAM)
Block 0
Block 1: ① years, ① ago, ① our, ② fathers
Block 2
Block 3: ③ brought
Block 4
Block 5
Block 6
Block 7: ① Four, ① score, ① and, ① seven
Block 8
① ① ③ ① ③ ①

Block Table

Physical block number	# filled
①7	①4
①1	①3 → 4②
③3	③1
–	–

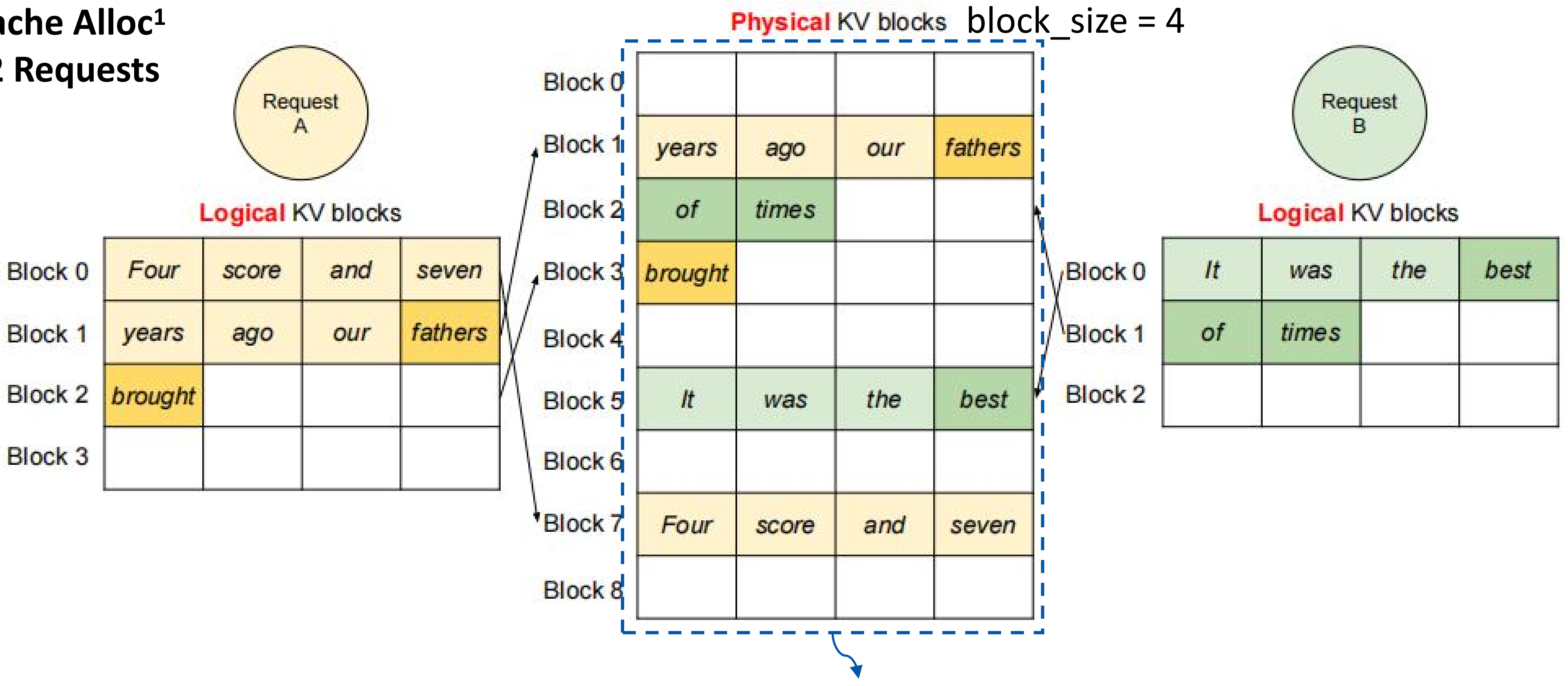
1. Figure source: https://zhuanlan.zhihu.com/p/691038809.
2. Kwon, Woosuk, et al. "Efficient memory management for large language model serving with pagedattention." Proceedings of the 29th symposium on operating systems principles. 2023.

Background: Paged Attention (2)

PagedAttention

KVCache Alloc[1]

for 2 Requests

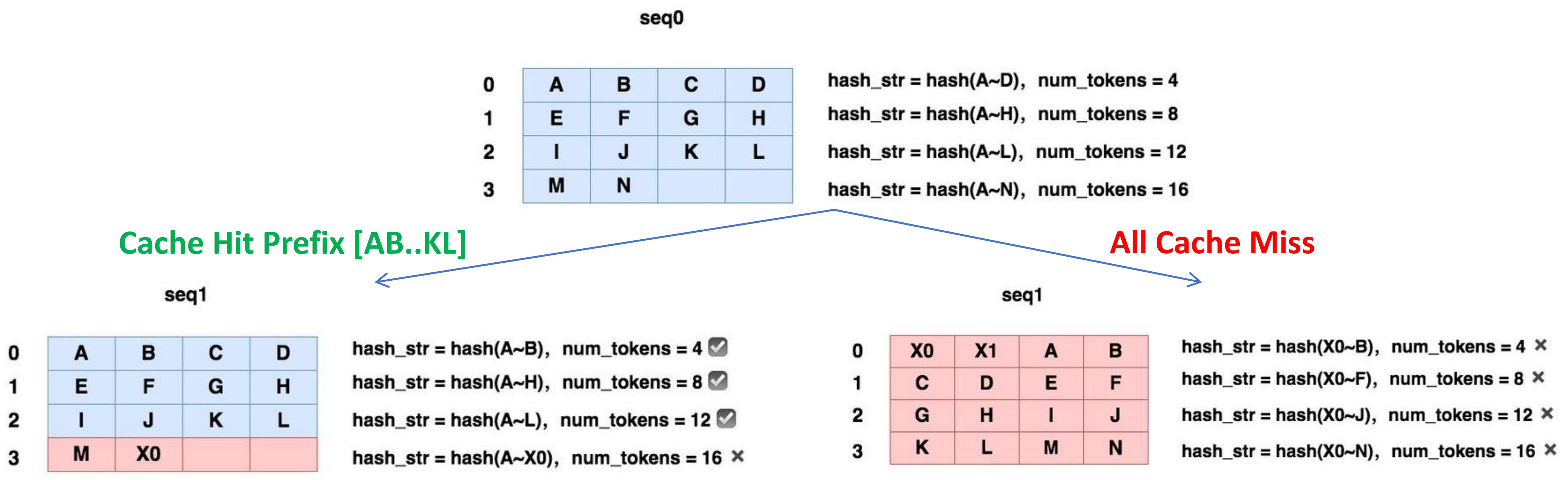


Available_KV_Cache_Memory

1. Kwon, Woosuk, et al. "Efficient memory management for large language model serving with pagedattention." Proceedings of the 29th symposium on operating systems principles. 2023.

Background: Prefix Caching

- **Cached page pool**
  - all $Available\_KV\_Cache\_Memory$ **not** allocated to running requests
  - **Add Page**: immediately after paged are freed
  - **Cache Eviction**: Free queue & LRU
  - **Cache Hit**: Determine the prefix of the new request that skip the recomputation of prefill



seq0

0	A	B	C	D	hash_str = hash(A~D), num_tokens = 4
1	E	F	G	H	hash_str = hash(A~H), num_tokens = 8
2	I	J	K	L	hash_str = hash(A~L), num_tokens = 12
3	M	N			hash_str = hash(A~N), num_tokens = 16

Cache Hit Prefix [AB..KL]

seq1

0	A	B	C	D	hash_str = hash(A~B), num_tokens = 4 ✓
1	E	F	G	H	hash_str = hash(A~H), num_tokens = 8 ✓
2	I	J	K	L	hash_str = hash(A~L), num_tokens = 12 ✓
3	M	X0			hash_str = hash(A~X0), num_tokens = 16 ✕

All Cache Miss

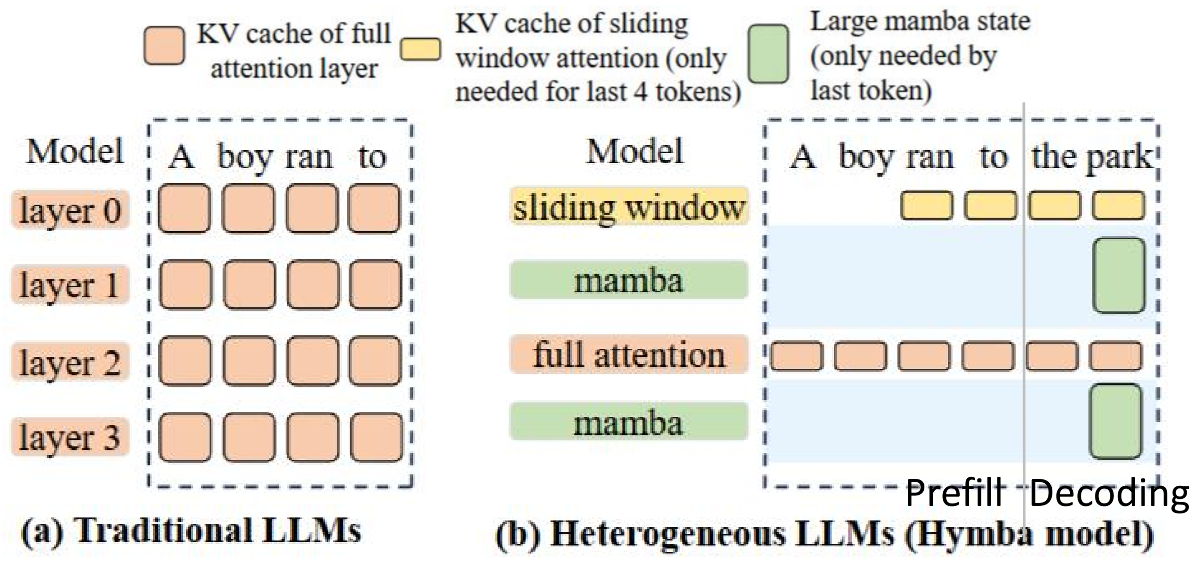
seq1

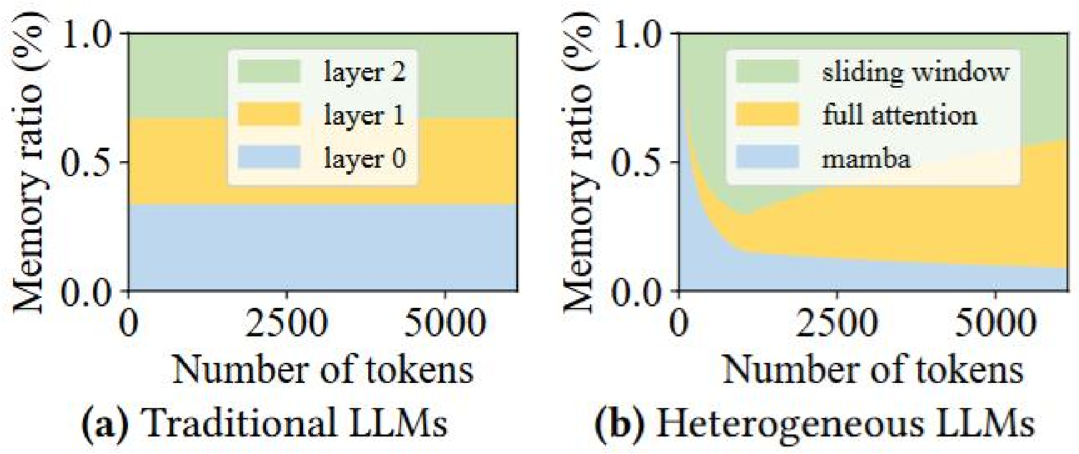
0	X0	X1	A	B	hash_str = hash(X0~B), num_tokens = 4 ✕
1	C	D	E	F	hash_str = hash(X0~F), num_tokens = 8 ✕
2	G	H	I	J	hash_str = hash(X0~J), num_tokens = 12 ✕
3	K	L	M	N	hash_str = hash(X0~N), num_tokens = 16 ✕

1. Figure source: https://zhuanlan.zhihu.com/p/707228704.

Background: Heterogeneous LLMs (1)

- **Heterogeneous LLM architecture**
  - Different attention mechanism **across layers**
  - Models often have embeddings (e.g. KV Cache) with different sizes
  - Example: *NVIDIA Hymba model*[1]
    - Sliding Window (SW) Layer: only attend to a sliding window of tokens
    - Mamba Layer[2]: use a large, fixed-size tensor to capture the information of all tokens


KV cache of full attention layer
KV cache of sliding window attention (only needed for last 4 tokens)
Large mamba state (only needed by last token)
Model
A boy ran to
layer 0
layer 1
layer 2
layer 3
(a) Traditional LLMs
Model
A boy ran to the park
sliding window
mamba
full attention
mamba
Prefill
Decoding
(b) Heterogeneous LLMs (Hymba model)

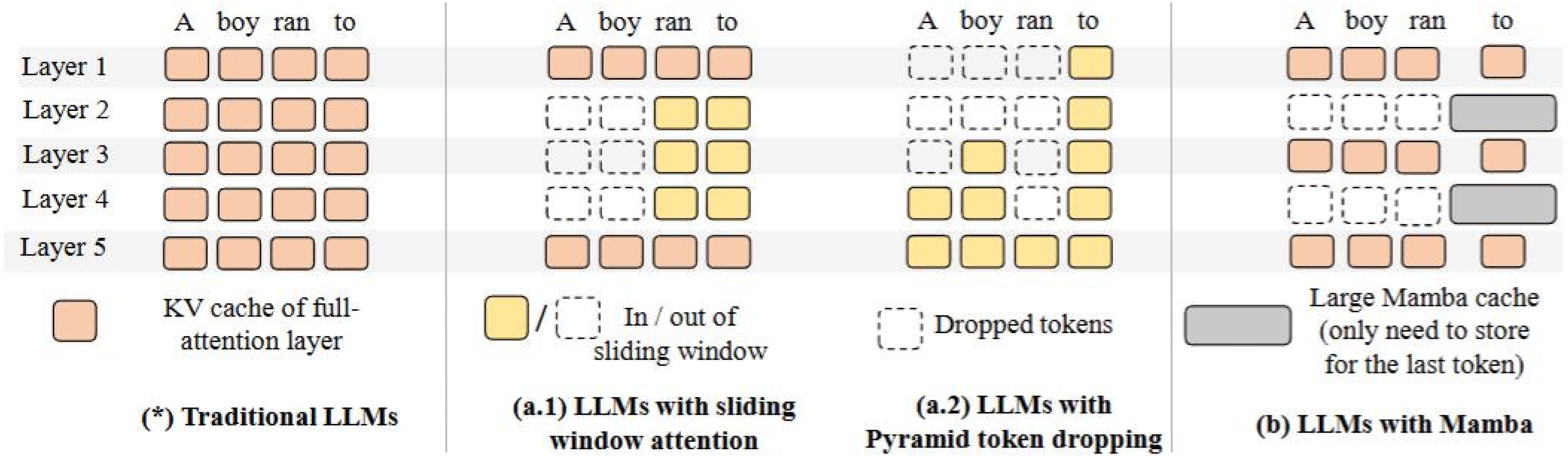


For shorter request: Mamba layers dominate mem.
For longer request: Full-att. layers dominate mem.

1. Hymba Hybrid-Head Architecture Boosts Small Language Model Performance. Nov. 2024. https://developer.nvidia.com/blog/hymba-hybrid-head-architecture-boosts-small-language-model-performance/
2. Mamba represents State space models, or linear attention. Mamda layers can viewed as SW layers with window_size 1.

Background: Heterogeneous LLMs (2)

- **Heterogeneity comes from new types of attention:**
  - **1) Sparse attention**:
    - Type 1: (mixed) sliding-window attention (SWA), e.g. Gemma-3 and Ministral
    - Type 2: token dropping, e.g. Pyramid
  - **2) Linear attention**: a tensor to capture the information of all tokens
    - Examples: Kimi Linear[1]



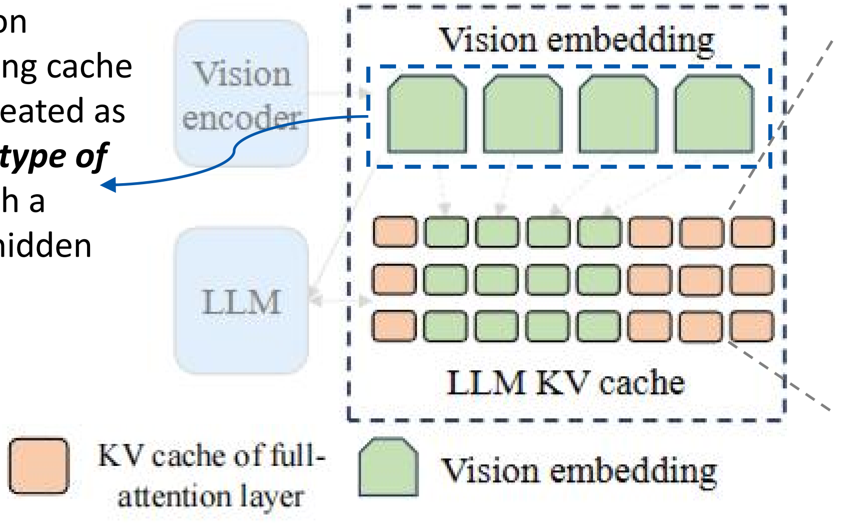
A boy ran to
Layer 1
Layer 2
Layer 3
Layer 4
Layer 5
KV cache of full-attention layer
(*) Traditional LLMs
In / out of sliding window
(a.1) LLMs with sliding window attention
Dropped tokens
(a.2) LLMs with Pyramid token dropping
Large Mamba cache (only need to store for the last token)
(b) LLMs with Mamba

1. Ping Gong (Presenter), and Xin Ren (Presenter). "Kimi Linear: An Expressive, Efficient Attention Architecture." ADSL Reading Group 2025 Fall, 2 Dec. 2025, adsl-rg.github.io/2025fall.html.

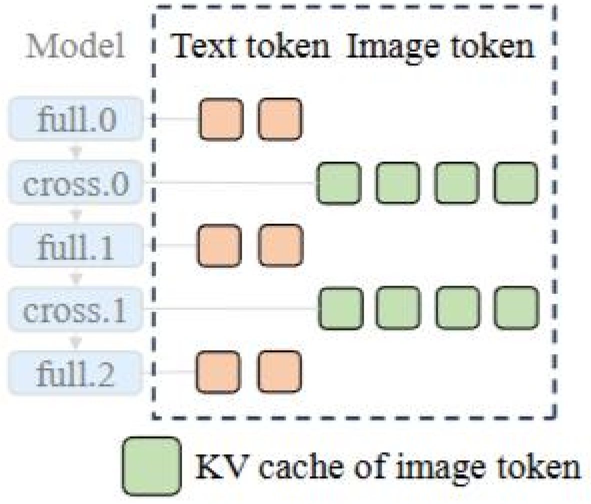
Background: Heterogeneous LLMs (3)

- **Heterogeneity comes from new types of attention:**
  - **3) VLMs**: {Images, texts} -> texts, e.g. Llama 3.2 Vision
    - Vision Embedding Cache. Vision Embedding also called image token.
    - Text Full Attention Layer: text_hidden -> text_hidden
    - Text-image Cross Attention Layer: {**Q_text_hidden**, **K_image**, **V_image**} -> text_hidden
  - **4) An Engine serving multiple models**: Speculative decoding

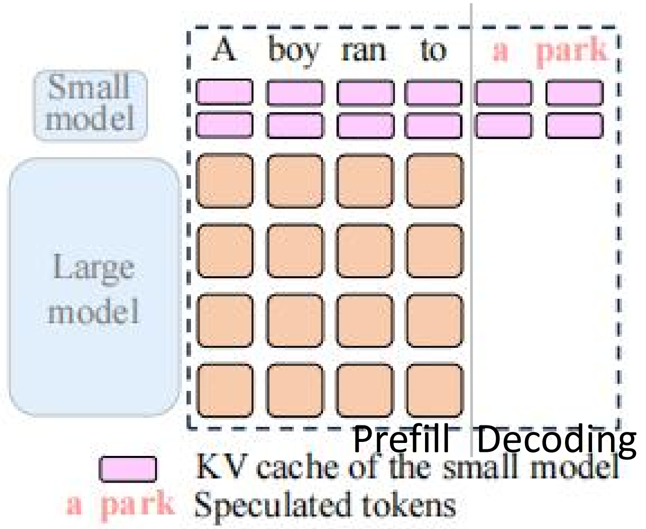
"the vision embedding cache can be treated as ***another type of layer*** with a specific hidden size"



(c.1) Vision language model



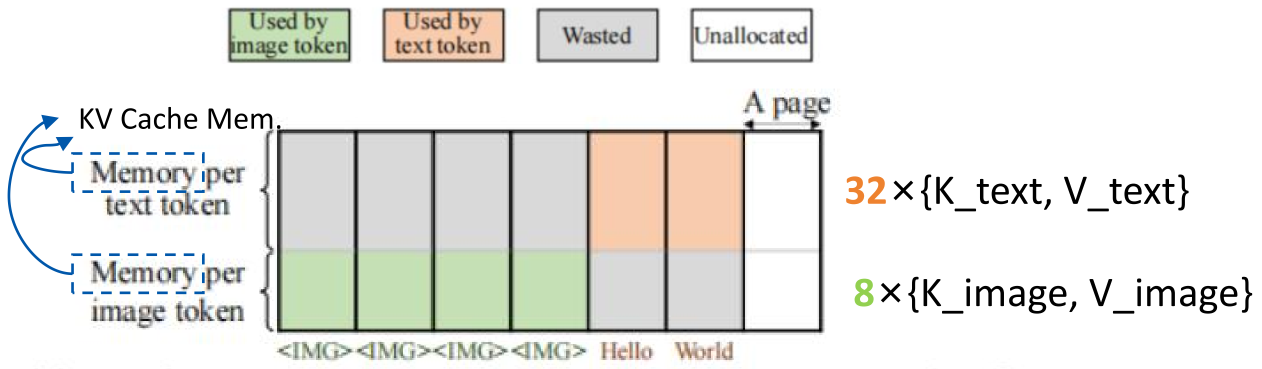
(c.2) Vision language model (cross attention based)



(d) Speculative decoding with two different-size models

Limitations of PagedAttention (1)

- **Analyzes PagedAttention's fragmentation**
  - For simplicity, set *tokens_per_page* = 1 (Block size = 1)
  - Llama 3.2 11B Vision: 32 full attention layers + 8 cross attention layers
    - Thus, one KV_cache_size of text_token = 4× KV_cache_size of image_token
  - Image tokens: vision encoder that takes images as input and generates vision embeddings.

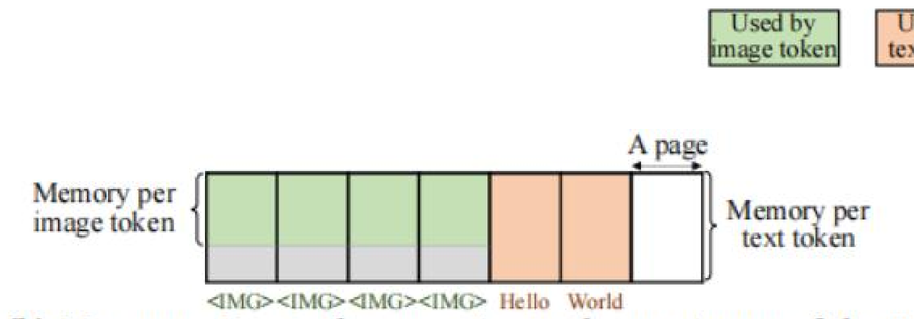


(a) PagedAttention. It always reserves the memory space for all tokens and for all layers, thus wastes memory when the token does not need to store KV cache for that layer.

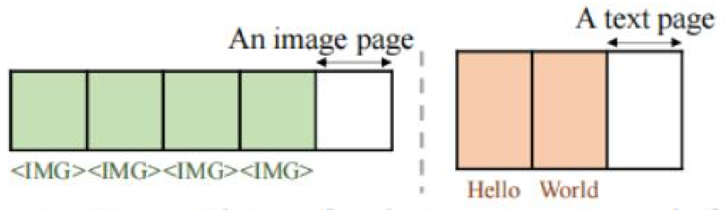
- Request Length = text_tokens# + image_tokens#
  - text_tokens#: T
  - image_tokens#: I
- KV Cache per layer per token: E bytes
- Policy of PagedAttention:
  - Demand: T*32*E + I*8*E
  - Allocated: $\underbrace{(32*E + 8*E)}_{\text{one } kv\_cache\_slot\_size}$ * (T+I)

Limitations of PagedAttention (2)

- **Max Page**: Allocated max(32*E, 8*E) * (T+I), still wastes.
- **Static Partition**: cannot adapt to dynamic workload changes
  - different ratio of text tokens (Full Att. Layers) and image tokens (Cross Att. Layers) in VLMs
    - This can be solved by analysing model in initialization[1].
  - different request length (text_tokens# + image_tokens#) in sparse attention



(b) Max page. It sets the page size to the maximum of the 2 types, thus wastes some memory for the smaller image tokens.



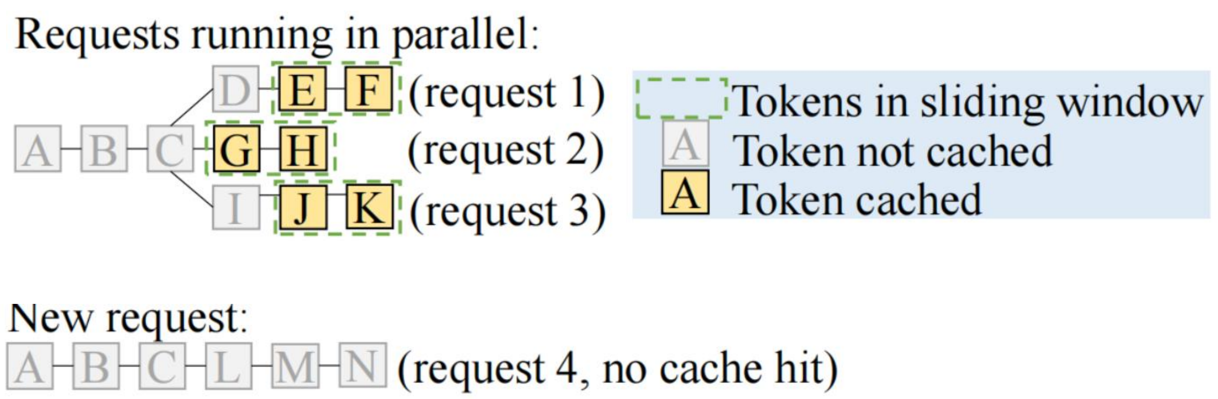
(c) Static partition with two fixed-size memory pools for image and text tokens. It works well on specific workloads, but cannot adapt to other token count like 2 image tokens plus 4 text tokens.

different request length

1. I guess.

Limitations of Prefix Caching

- **SWA (sliding window Attention) Model**: two types layers
  - Full-attention layer
  - Sliding window layer
- **Existing Prefix Caching in SWA Model**
  - Only caching tokens inside sliding window
  - Maximizes batch size √
  - Causes cache miss for common tokens ×



Just Caching **all layers {K, V}**: EF, GH, JK
The decoding will have more free slot

All cache missing!
If cached **full-att. layer {K, V} of ABC**, it will hit!

Overview of Jenga

- KV Cache of 1 token = ΣL_type1{K, V} + ΣL_type2{K, V} + ... + ΣL_typen{K, V}
- 1 Small Page = ΣL_type{K, V}
- Two-level
  - LCM allocator
  - Layer-specific allocator

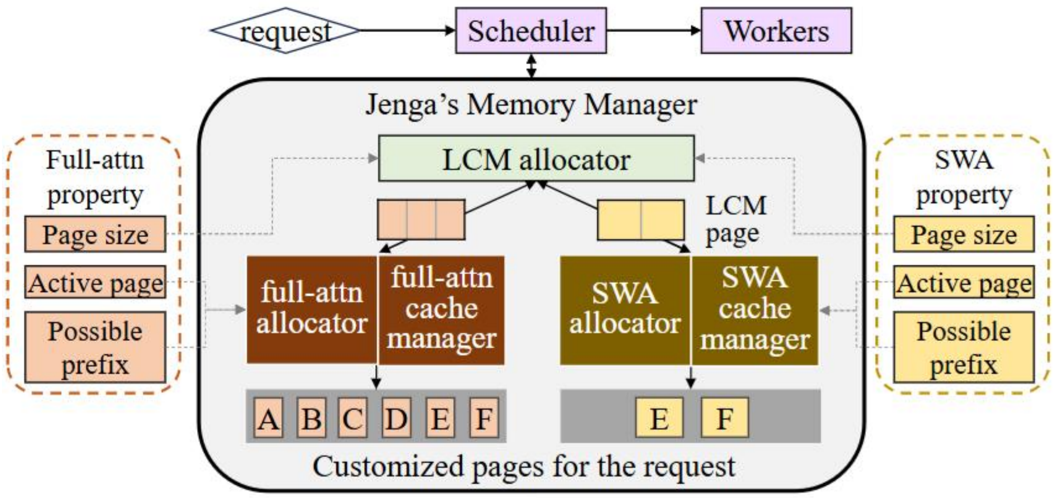
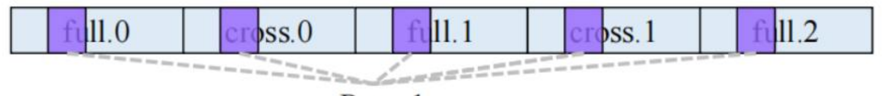
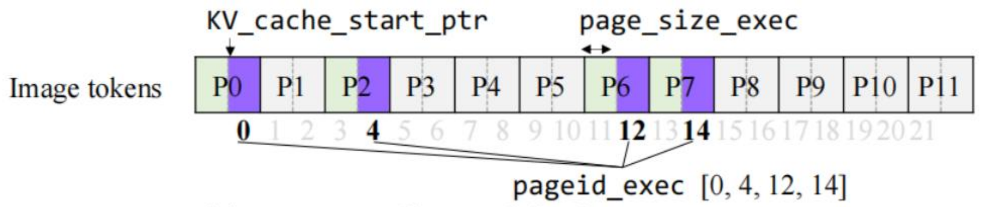


Figure 7. Overview of JENGA



(a) Memory layout of PagedAttention

- First, partitions memory into **<u>layers</u>**
- Then, partitions each **<u>layer</u>** into **<u>pages</u>**
- vLLM, SGLang, TGI, FlashAttention, FlashInfer



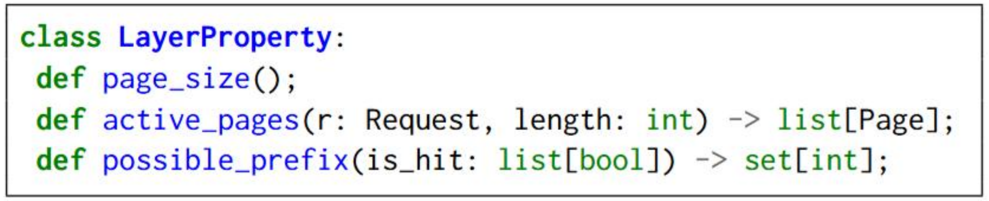
(c) Memory allocated for layer cross.1

- First, partitions the memory into **<u>pages</u>**
- Then, partitions each **<u>page</u>** into **<u>layers</u>**

Overview of Jenga

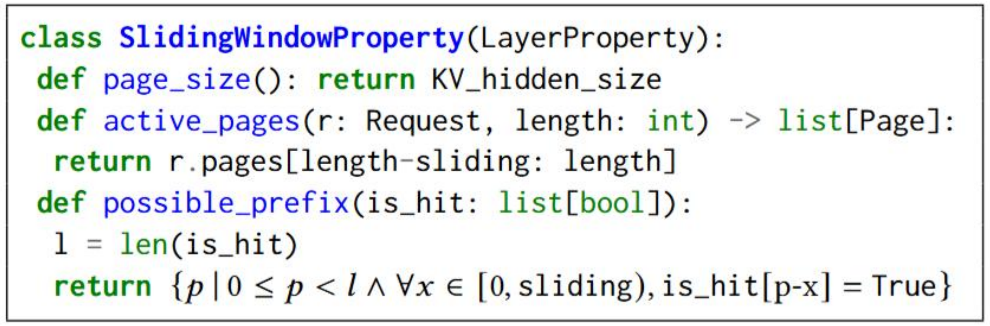
- Unified ***LayerProperty* Interface** for a Request
  - **Page size:** $KV\_size\_per\_token$ in type of layer
  - **Active_pages:** used to compute future token
  - **Possible_prefix:**
    - input: a boolean list marking which pages are cached
    - output: return all valid prefixes of that layer

- **Sliding Window Layer** as an example
  - Active_pages:
    - tokens in sliding window
  - Possible_prefix:
    - caching sliding_window_size tokens



```
class LayerProperty:
 def page_size();
 def active_pages(r: Request, length: int) -> list[Page];
 def possible_prefix(is_hit: list[bool]) -> set[int];
```

(a) Layer properties that JENGA is aware of



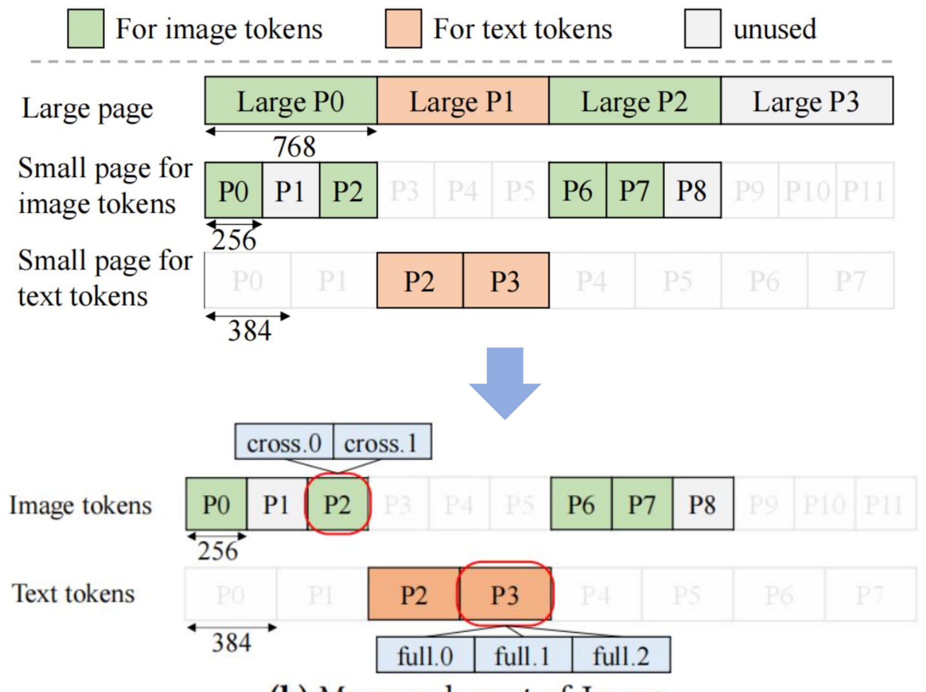
```
class SlidingWindowProperty(LayerProperty):
 def page_size(): return KV_hidden_size
 def active_pages(r: Request, length: int) -> list[Page]:
  return r.pages[length-sliding: length]
 def possible_prefix(is_hit: list[bool]):
  l = len(is_hit)
  return {p | 0 ≤ p < l ∧ ∀x ∈ [0, sliding), is_hit[p-x] = True}
```

(b) Example implementation of sliding window layer

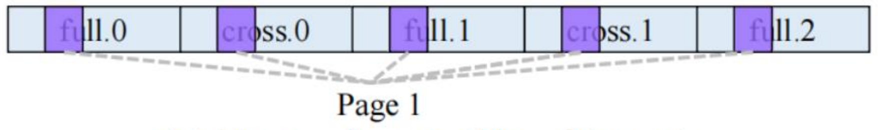
LCM Allocator & Memory Layout

NORTHWESTERN POLYTECHNICAL UNIVERSITY

- **Assume Llama vision model = 3 full attention layers + 2 cross attention layers**
  - *KV_size_per_token* of each layer is 128
  - *KV_size_per_text_token* 128×3 = 384, *KV_size_per_image_token* 128×2 = 256
  - LCM (256, 384) = 768
- **Request**: <IMG><IMG><IMG><IMG>Hello world

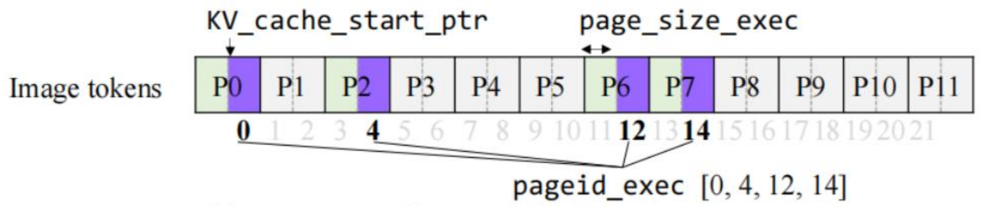


(b) Memory layout of JENGA



(a) Memory layout of PagedAttention

- First, partitions memory into **layers**
- Then, partitions each **layer** into **pages**
- vLLM, SGLang, TGI, FlashAttention, FlashInfer



(c) Memory allocated for layer cross.1

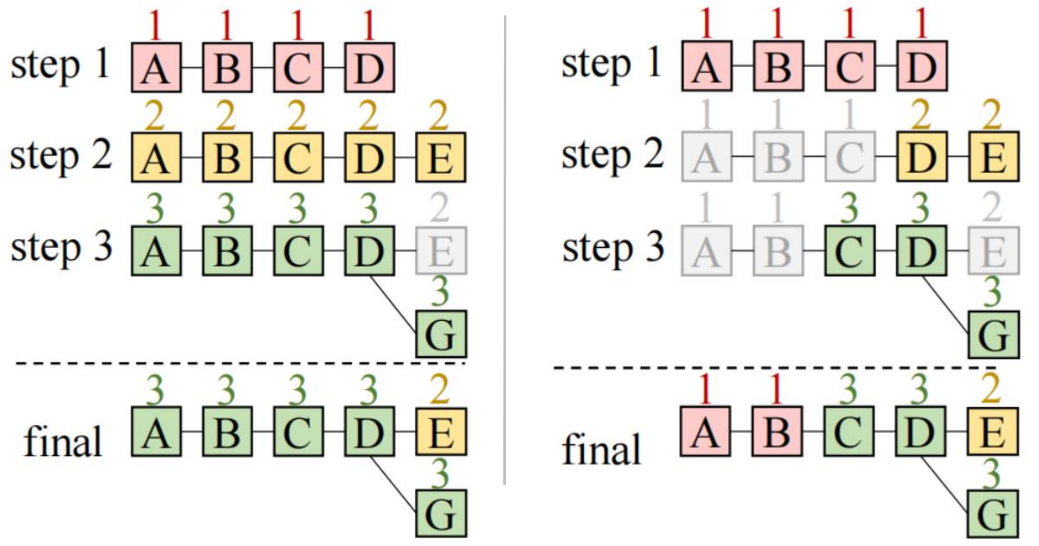
- First, partitions the memory into **pages**
- Then, partitions each **page** into **layers**

Customized Cache Eviction

- **Evict the page (~~whole~~layer {K, V}) of a token**
  - Prioritize eviction for unimportant pages
    - e.g., pages outside sliding window
  - Balanced eviction across layers
- **Update last_access_time of**
  - active_pages (e.g., pages in sliding window)
  - pages for saving the generated KV cache
- **Evict the the least last_access_time**

Step	Request	Full attention	Sliding window	
1	1's prefill	ABCD->E	ABCD->E	SW_size=4
2	1's decode	ABCDE->F	DE->F	SW_size=2
3	2's prefill	ABCDG->H	CDG->H	SW_size=3

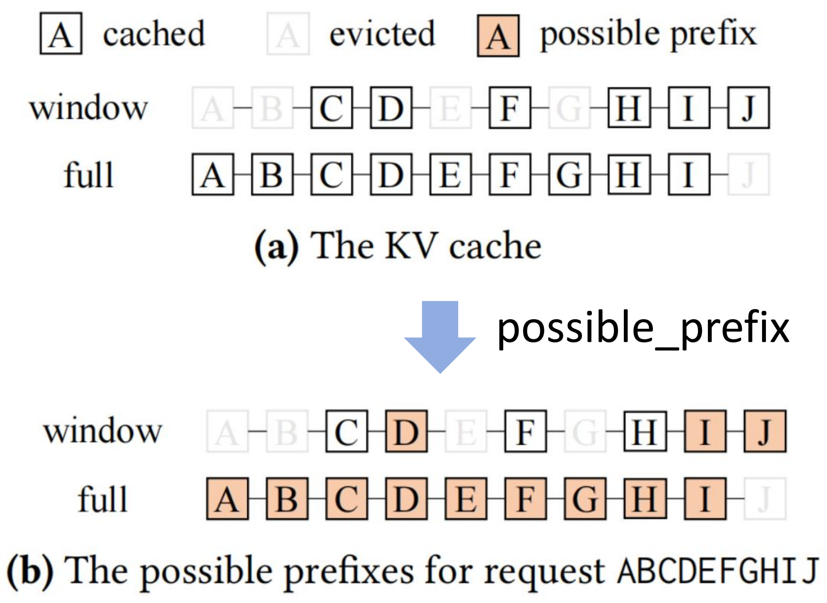
Note that the generated token does not have KV cache.



(b) Full-attention layer (c) Sliding window layer

Customized Cache Hit

- **Cache hit rules differ across layer types**
  - possible_prefix: All valid prefixes of a layer for a request
  - get_possible_prefix for each layer type to identify valid prefixes.
  - (KV) Cache hit prefix: the longest common prefix valid across all layers.



(b) The possible prefixes for request ABCDEFGHIJ

- **SW Layer**
  - possible_prefix returns {4, 9, 10}
  - The possible prefixes: {ABCD, ABCDEFGHI, ABCDEFGHIJ}
- **Full attention Layer**
  - possible_prefix returns {1, 2, 3, ..., 9}
  - The possible prefixes in Full Layer: {A}, {AB}, ..., {ABCDEFGHI}.

- The cache hit prefix is **{ABCD}, {ABCDEFGHI}**.

More Optimization

- **Customization for Different Layers**
  - Sliding Window Layer
  - Mamba Layer
    - only caches the state of every 512 tokens
    - active_pages only returns the page of the last cached token
  - Local Attention (e.g., Llama 4)
    - a request is divided into 8192-token chunks
    - active_pages includes the pages belonging to the same chunk
  - Vision Embedding Cache and Vision Cross Attention Cache
    - evict all tokens from one image
    - active_pages include all pages of the same image
- **Common Page Pool**: future imporve cache hit rate
  - alongside the regular cached page pool, separately.

Evaluation: Setup

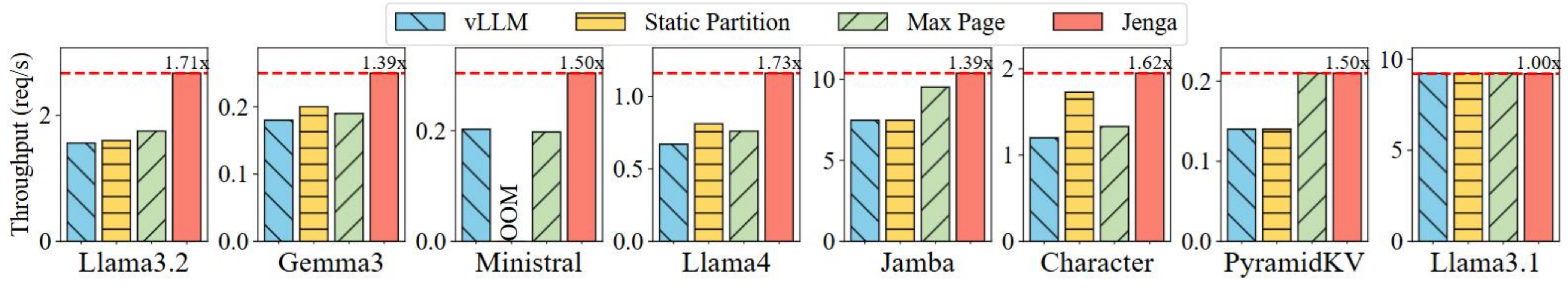
- **Testbed 1**: one NVIDIA H100 80GB GPU, 2 Intel Xeon Platinum CPUs, CUDA 12.4
- **Testbed 2**: one NVIDIA L4 24GB GPU, 2 AMD EPYC 7F52 CPUs, CUDA 12.4
- **Model**:
  - Llama 3.2 vision with cross attention layers
  - Gemma-3, Ministral and Character.ai with sliding window layers
  - Llama 4 with local attention layers
  - Jamba-1.5 with Mamba layers
  - PyramidKV drops some tokens
  - Llama 3.1: Traditional Full attention model
- **Dataset**:
  - MMLU-pro for text-only models (length <= 3076)
  - MMMU-pro for multi-modality models
  - arXiv-QA, a long-context dataset
- **Baseline**: vLLM, SGLang, TGI

Model	Dataset	H100	H00-TP	L4
Llama 3.2 Vision	MMMU-pro	11B	1	11B*
Gemma-3	arXiv-QA	12B	1	4B
Ministral	arXiv-QA	8B	1	8B*
Llama 4	arXiv-QA	109B	8	OOM
Jamba-1.5	MMLU-pro	52B	4	OOM
character.ai	MMLU-pro	70B*	1	8B
PyramidKV	MMLU-pro	70B*	1	8B
Llama 3.1	MMLU-pro	70B	4	8B

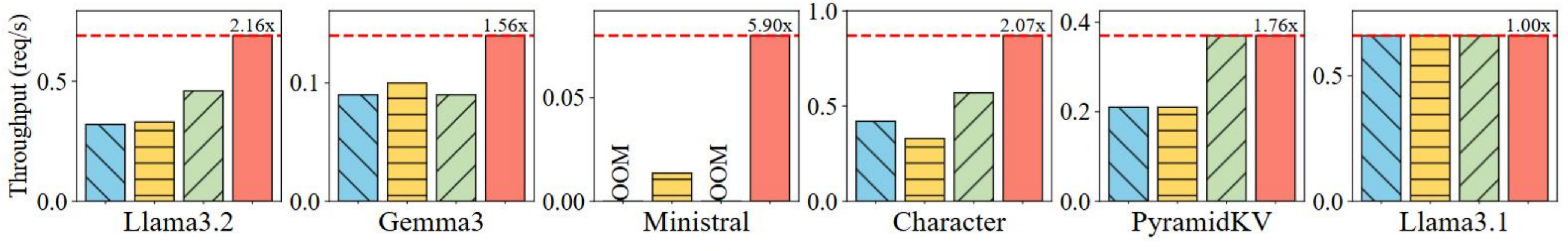
* means with FP8 quantization.

E1-1 End-to-end Thpt

- **vLLM**: traditional PagedAttention
- **Static Partition**: Partition KV cache memory by layer type. Each layer receives an equal number of pages.
- **Max Page**: All layers adopt a common page size equal to the largest among them



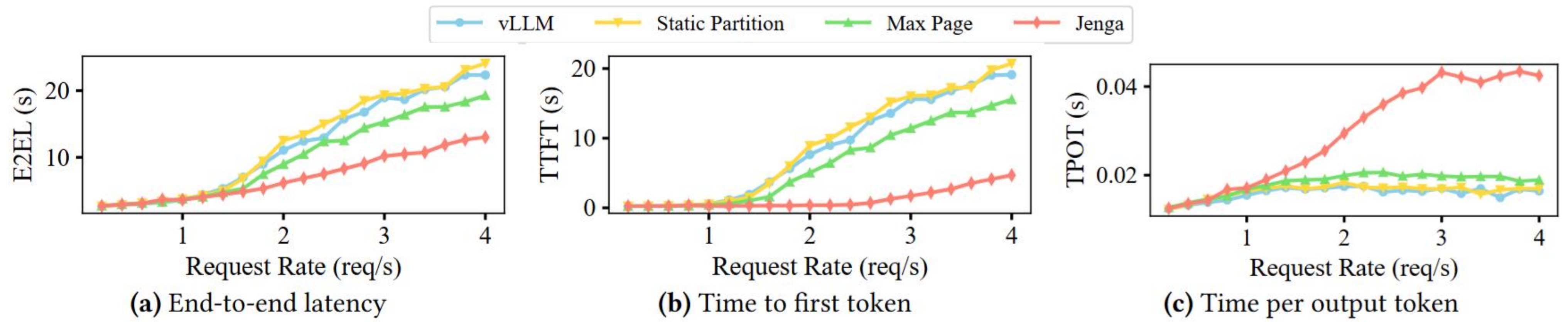
(a) H100 GPU



(b) L4 GPU

E1-2 End-to-end Latency

- **vLLM**: traditional PagedAttention
- **Static Partition**: Partition KV cache memory by layer type. Each layer receives an equal number of pages.
- **Max Page**: All layers adopt a common page size equal to the largest among them

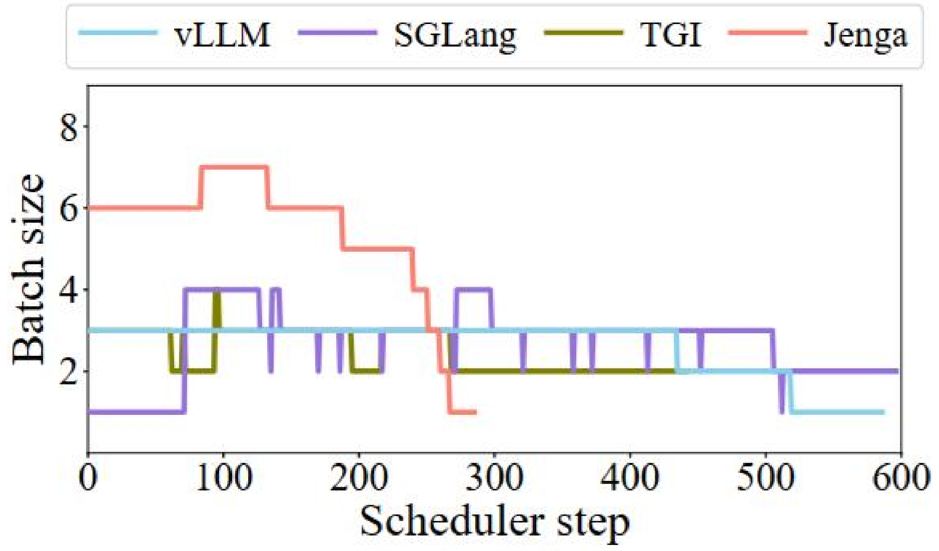


Averaged Latency for the **Llama Vision** Model

The TPOT of Jenga is larger than vLLM because Jenga **batches more requests** and **has more computation** in each step. (details in next slide) Jenga can achieve the same TPOT if scheduling the same number of requests in each step.

E2-1 Decode batch size

- Workload: simulates the typical long document question-answering
  - 20 requests arriving at the inference engine all at once
  - input length ~55-110 thousand tokens
  - output length ~50-100 tokens



Timeline of decode batch size for Ministral model

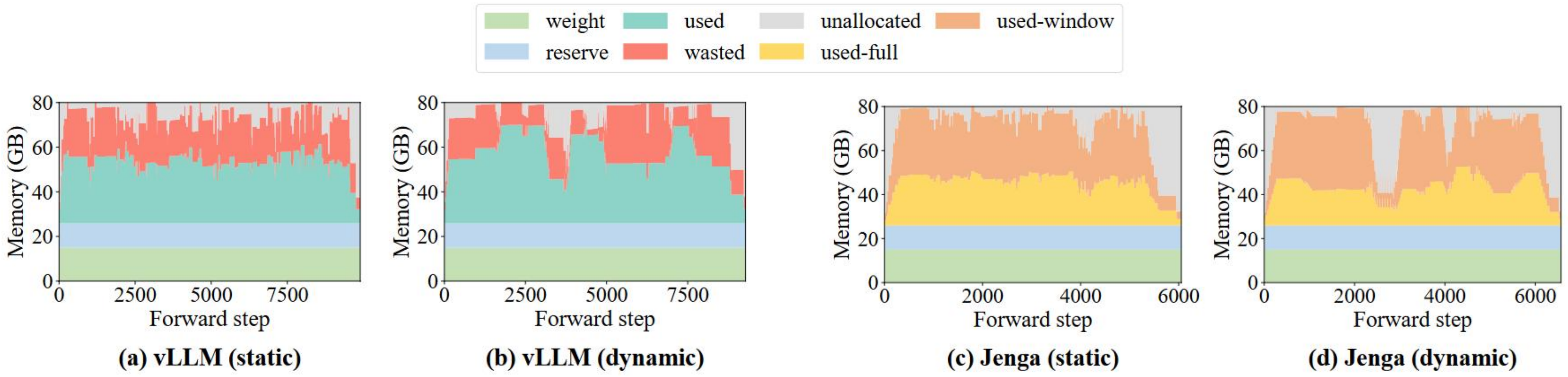
	Avg. Decode Batch Size
vLLM	2.63
SGLang	2.74
TGI	2.50*
Jenga	**5.39**

*TGI does not support the *--ignore-eos* flag (proposed in vLLM v0.6.0), and thus generates fewer tokens compared to the other inference engines.

E2-2 Fragmentation Analysis (Breakdown)

西北工業大學 NORTHWESTERN POLYTECHNICAL UNIVERSITY

- **Static trace**: the request length distribution does not change over time
- **Dynamic trace**: the average length forms a uniform distribution over time
- **Metrics**:
  - reserve: model activations and cuda graphs
  - wasted: memory that is allocated but not storing useful KV cache
  - unallocated
- **vLLM wastes 38.2% KV cache memory on average, while Jenga only has 0.04%**

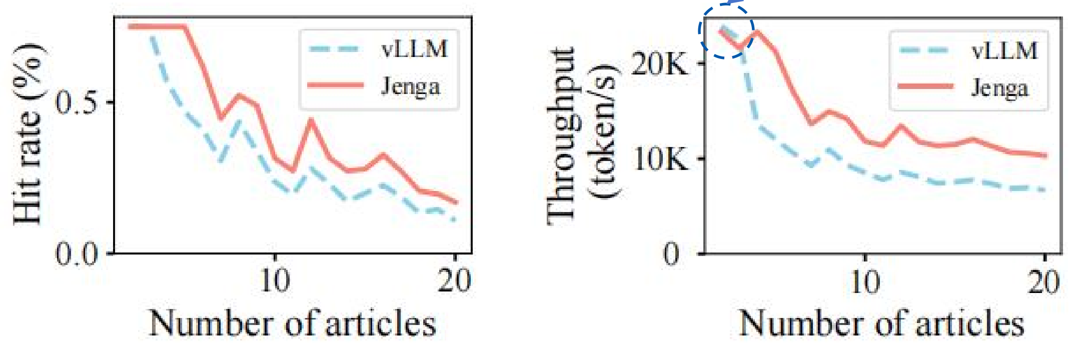


Timeline of memory usage for Ministral model

E2-3 Prefix Caching

- When the number of articles is small (e.g., <= 3), both systems can cache all articles.
- When the number of articles is big:
  - Jenga has up to **1.60×** cache hit rate as being able to prioritize the eviction of KV cache
  - The higher cache hit rate saves more computation, which birngs 1.77× throughput

the slight overhead of Jenga is Jenga needs to allocate memory twice



(a) Hit rate **(b)** Throughput

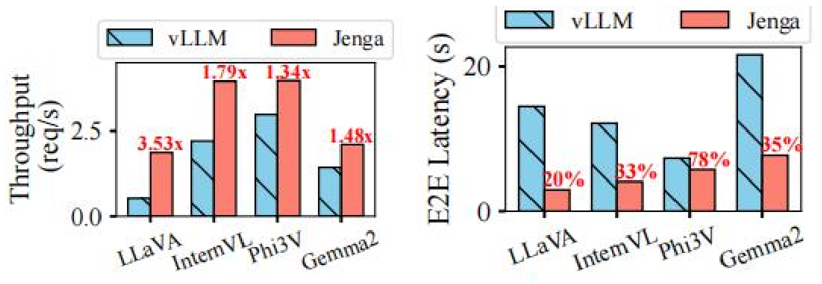
Prefix caching with different number of articles in the arXiv-QA dataset

	Full Attention	+Customized LRU	+Common Page Pool
arXiv-QA	2.4%	15.2%	34.6%
Mooncake	14.1%	15.5%	20.9%

Cache hit rates of prefix caching optimizations for **Gemma-3** model

E3 Case 1-5

- Case 1: VLM with Vision embedding cache
- Case 2: Speculative decoding
  - vLLM-max: using a uniform page size as in the PagedAttention
  - vLLM-manual uses a manually-designed memory allocation strategy for speculative decoding by SmartSpec
- Case 3: Multi-turn conversation
  - 500-token common prefix for all conversations
- Case 4: Chain-of-thought
  - the model produces long intermediate reasoning
- Case 5: Parallel sampling
  - generates multiple outputs per request



(a) Throughput (b) E2E latency

Figure 19. Vision language model with chunked prefill.

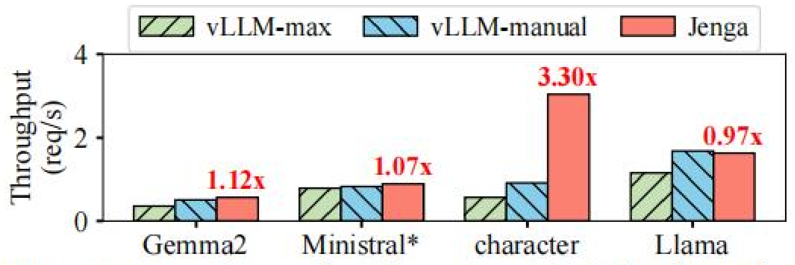
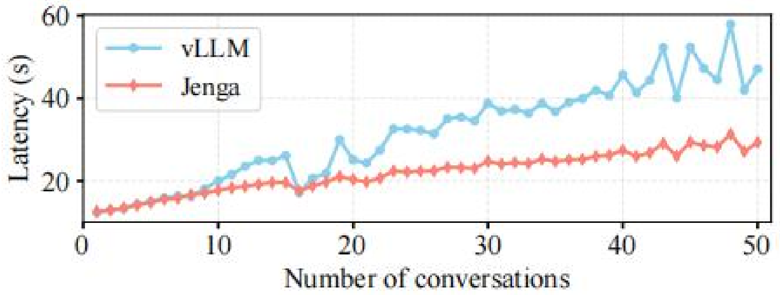
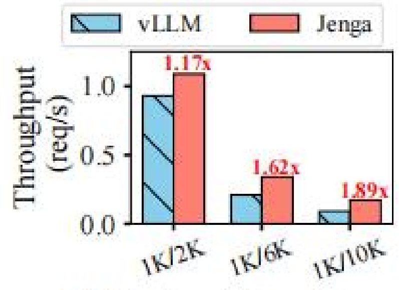


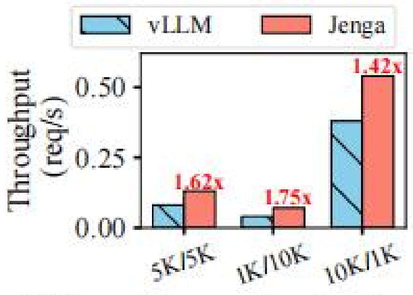
Figure 20. Speculative decoding. Amplified the throughput of Ministral by 10 × for better visualization.



(a) Multi-turn conversation

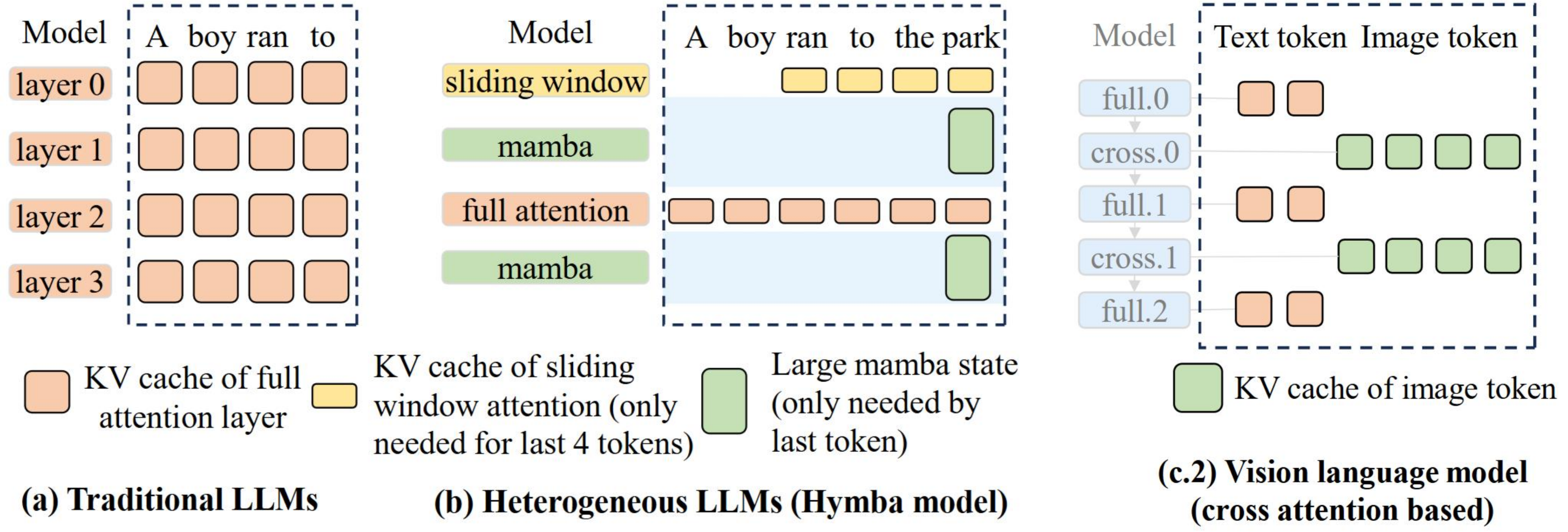


(b) Chain of thought



(c) Parallel sampling (n=3)

Summary


Model
A boy ran to
layer 0
layer 1
layer 2
layer 3
KV cache of full attention layer
(a) Traditional LLMs
Model
A boy ran to the park
sliding window
mamba
full attention
mamba
KV cache of sliding window attention (only needed for last 4 tokens)
Large mamba state (only needed by last token)
(b) Heterogeneous LLMs (Hymba model)
Model
Text token Image token
full.0
cross.0
full.1
cross.1
full.2
KV cache of image token
(c.2) Vision language model (cross attention based)

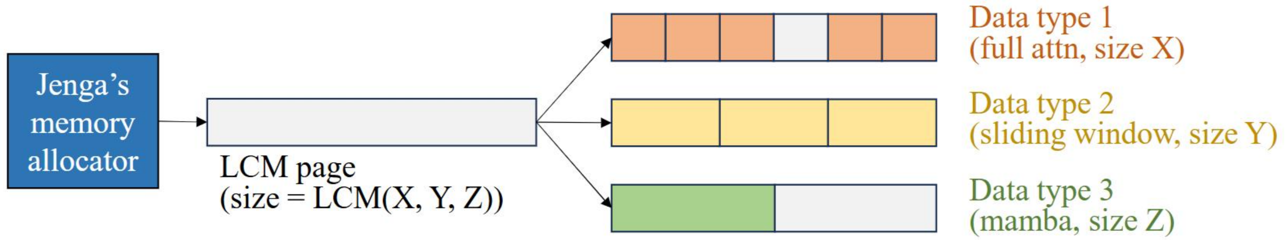
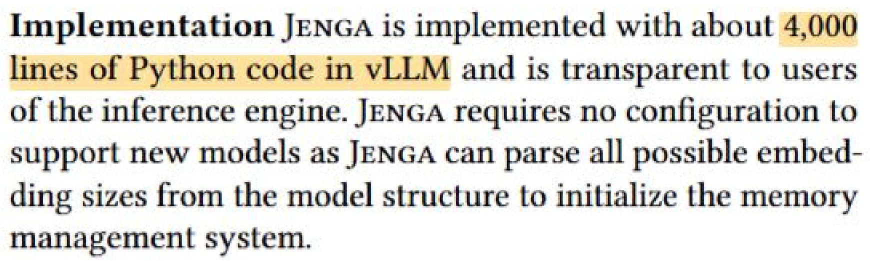

Jenga's memory allocator
LCM page (size = LCM(X, Y, Z))
Data type 1 (full attn, size X)
Data type 2 (sliding window, size Y)
Data type 3 (mamba, size Z)

Figure 3. LCM allocator in JENGA

LCM means *least common multiple*.



Implementation JENGA is implemented with about 4,000 lines of Python code in vLLM and is transparent to users of the inference engine. JENGA requires no configuration to support new models as JENGA can parse all possible embedding sizes from the model structure to initialize the memory management system.

西北工業大學
NORTHWESTERN POLYTECHNICAL UNIVERSITY

Thanks for Listening

Mingxuan Liu
PhD student at Northwestern Polytechnical University
January 6, 2026