vAttention: Dynamic Memory Management for Serving LLMs

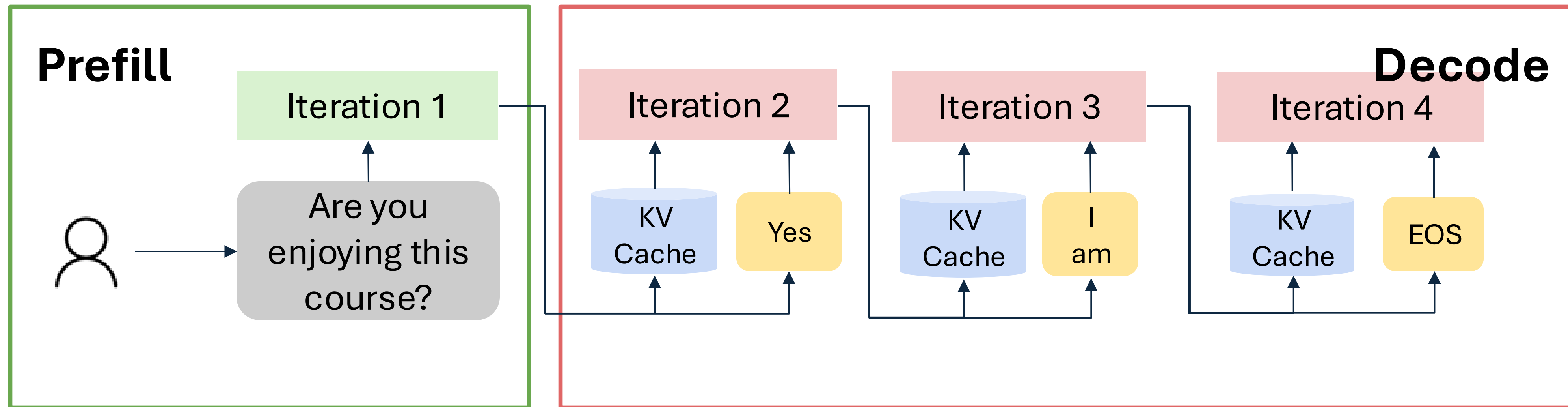
Presenters: Rahul Bothra, Chengyi Wang
11th October 2024

Our **comments** in purple boxes

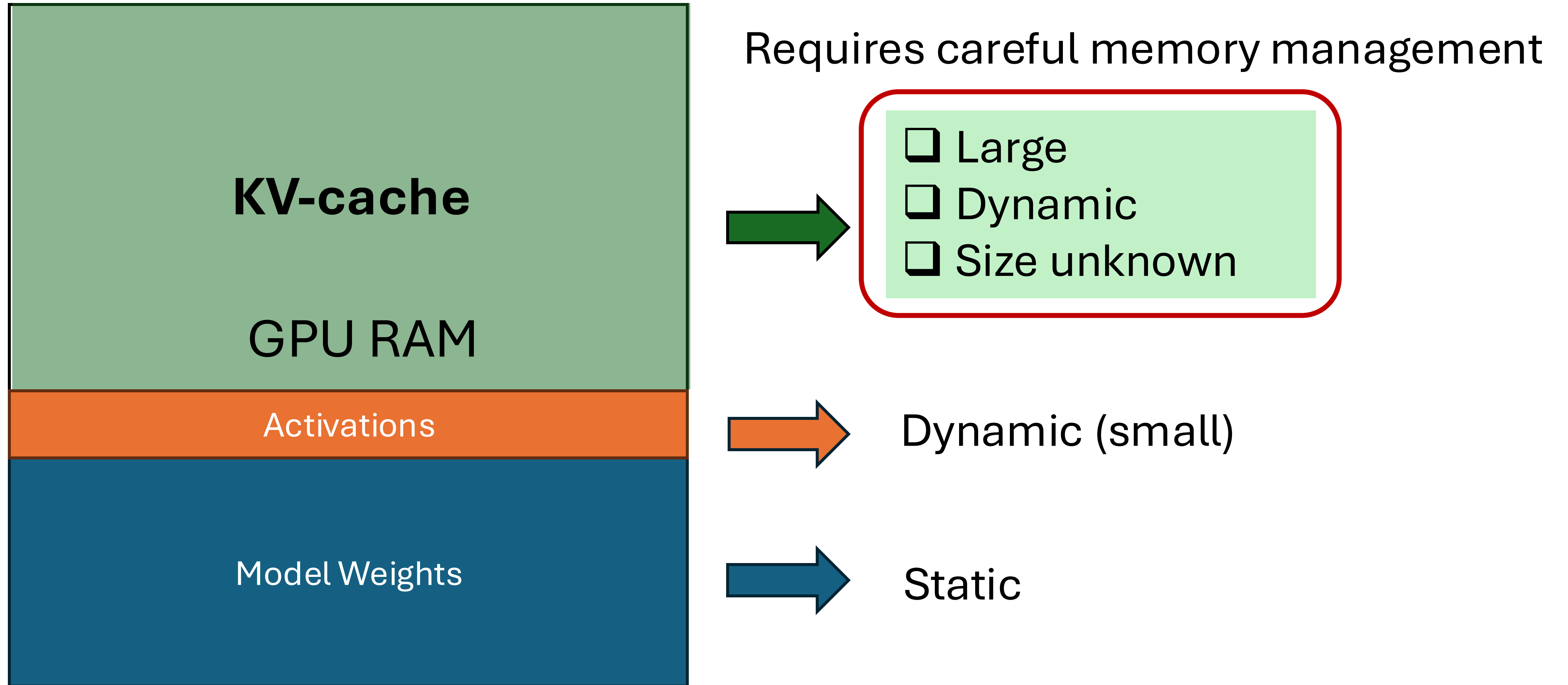
Contents

1. LLM Inference primer + KV Cache
2. Prior work on memory management (Orca and Paged Attention)
3. vAttention Design (Design and Challenges (CUDA paging size))
4. vAttention Evaluation
5. Analysis and Future work

LLM Inference Primer



LLM Inference memory footprint



KV Cache Memory Management

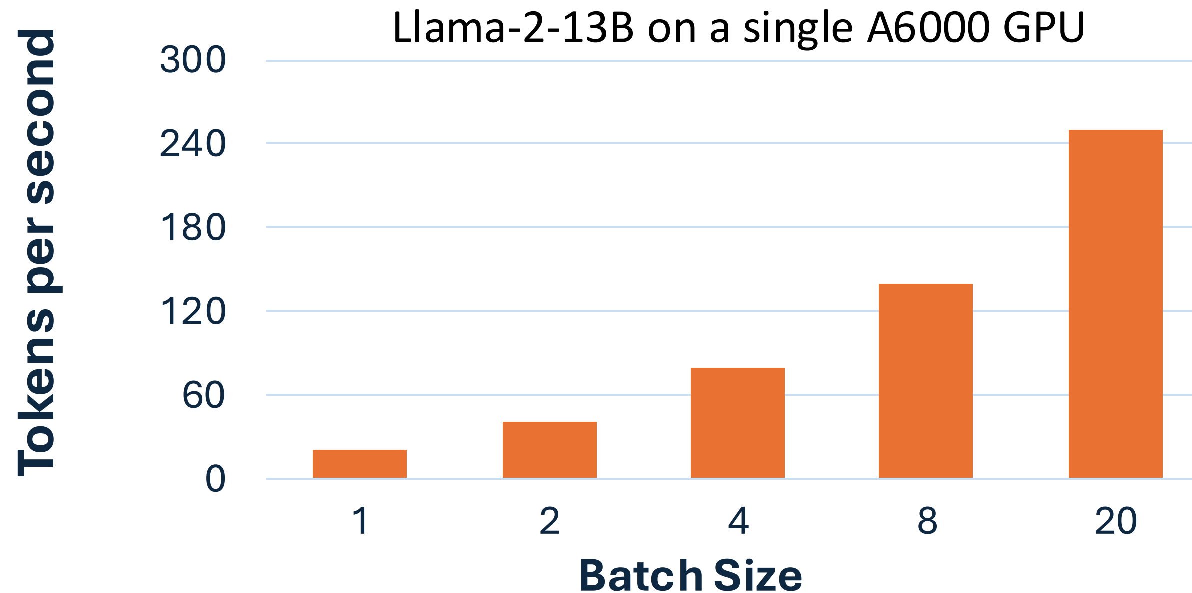
- KV-cache is **large, dynamic** and **size** is **unknown/variable**
- GPT-3: 1000 tokens = 4.5GB memory
- Grows one-token at a time (autoregressive decoding)
- Don't know request lengths in advance

Why care about memory management?

Comment: Predicting the final KV cache size in advance could be useful.

Why is memory management important?

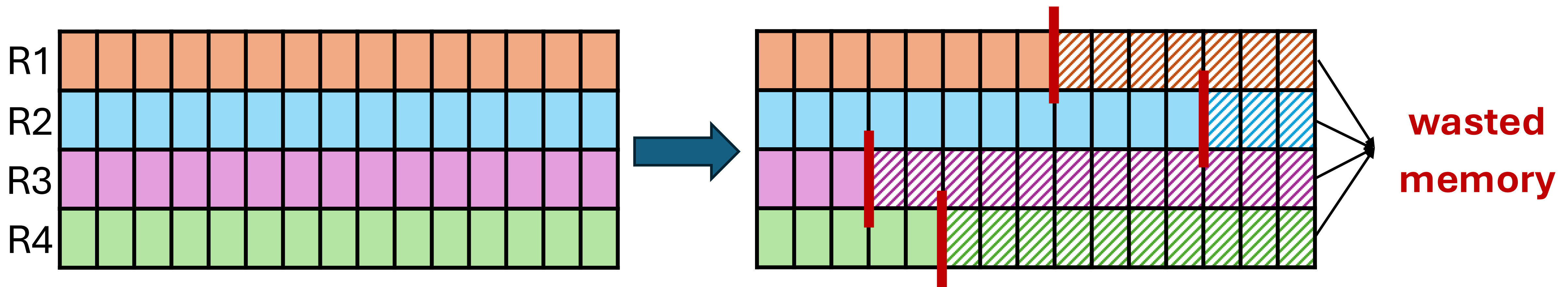
- LLM Inference throughput depends on batch size



- Batch size depends on memory (KV-cache) allocator

A simple KV Cache Memory Manager

- **Assume** length(R_i)== max context length
- Allocate all memory upfront
  - e.g., max model length for GPT-3 = 4K
  - allocate 18GB memory for each request (= 4*4.5GB)



Can’t serve more requests (though memory is underutilized)

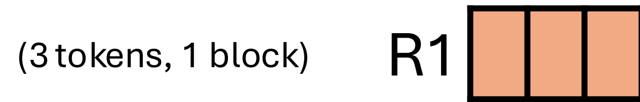
A better approach – vLLM (SOSP '23)

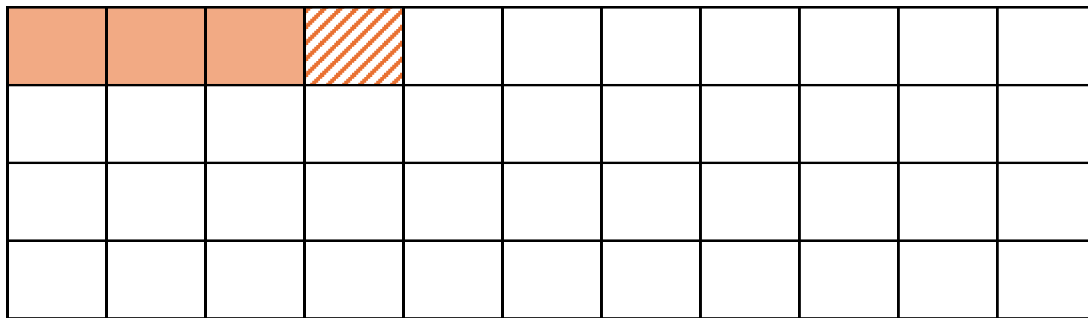
- Dynamic fine-grained memory allocation for KV-cache

- **Dynamic:**
  - On demand allocation

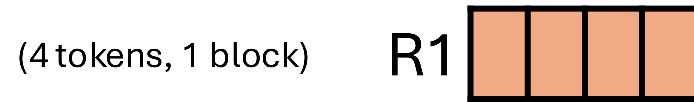
- **Fine-grained:**
  - Divide memory into fixed-size blocks (e.g., 16 tokens)
  - Allocate one block at a time

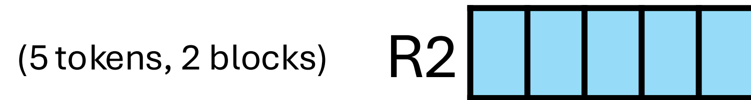
Memory Allocation with vLLM

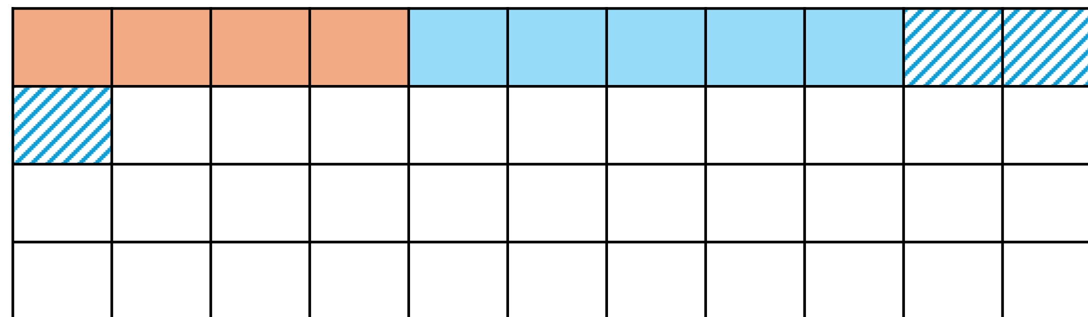




Memory Allocation with vLLM

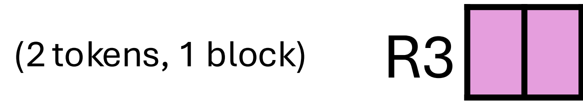


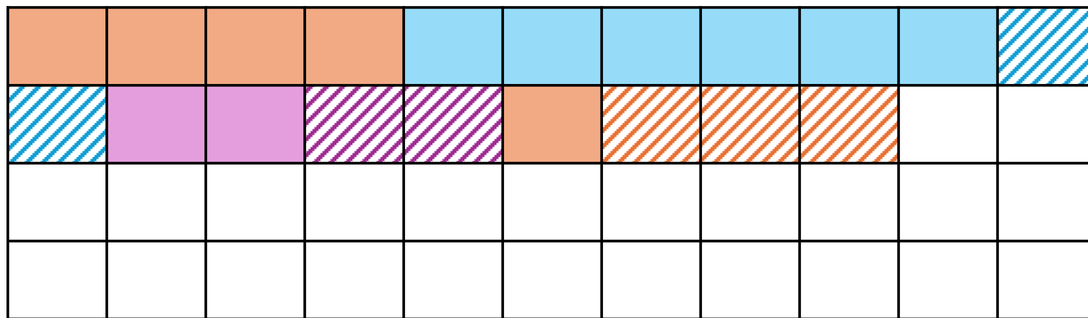




GPU Memory (block size = 4)

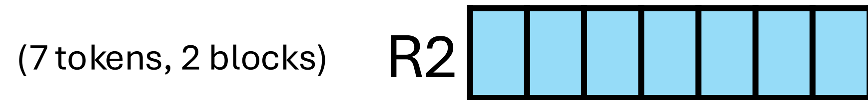
Memory Allocation with vLLM

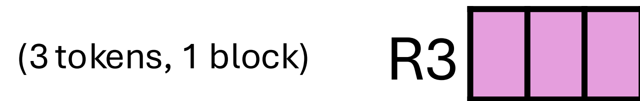


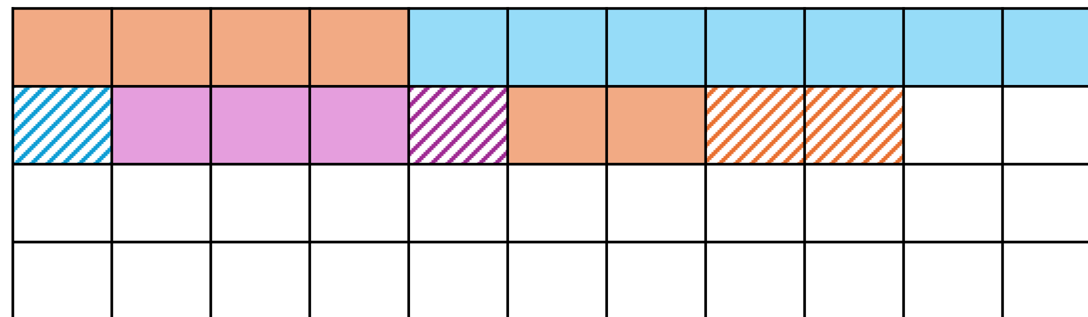


GPU Memory (block size = 4)

Memory Allocation with vLLM

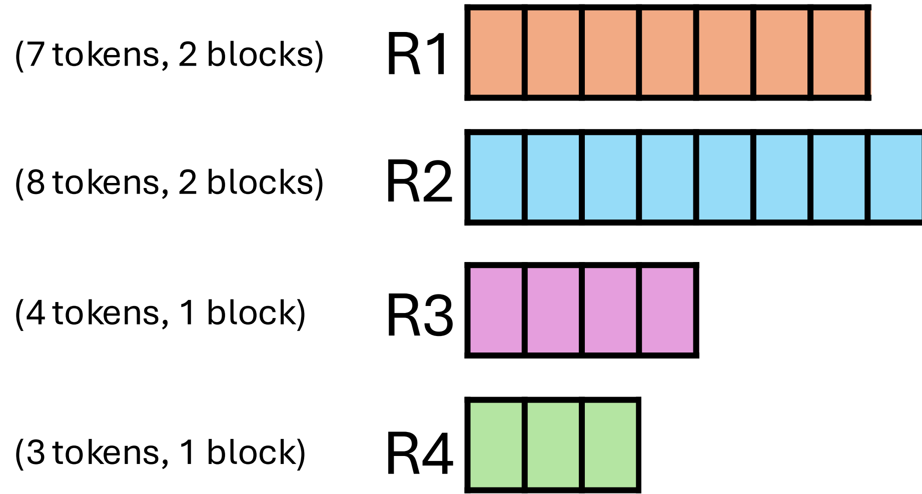


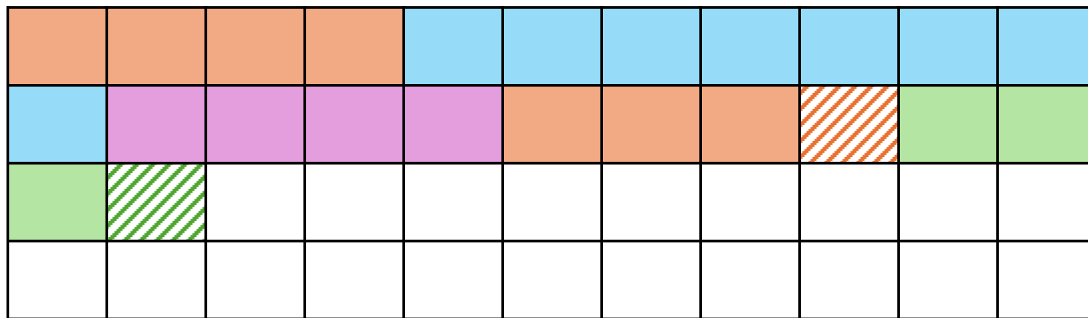




GPU Memory (block size = 4)

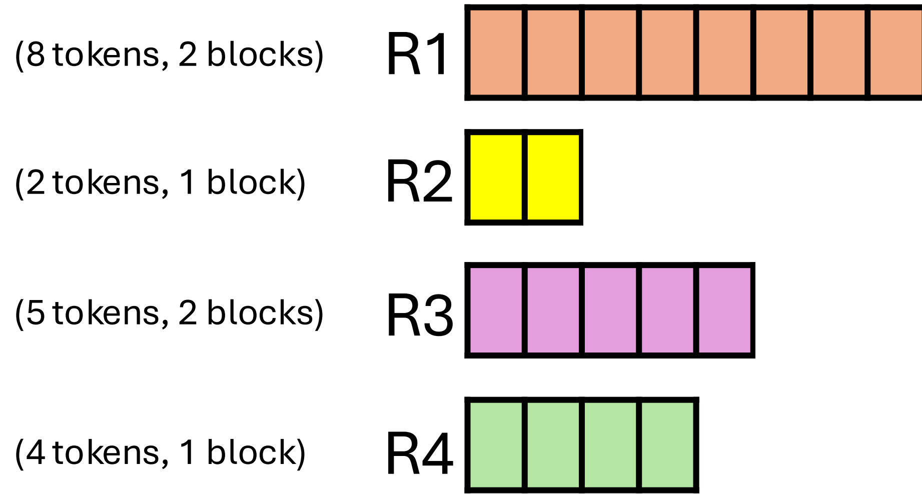
Memory Allocation with vLLM

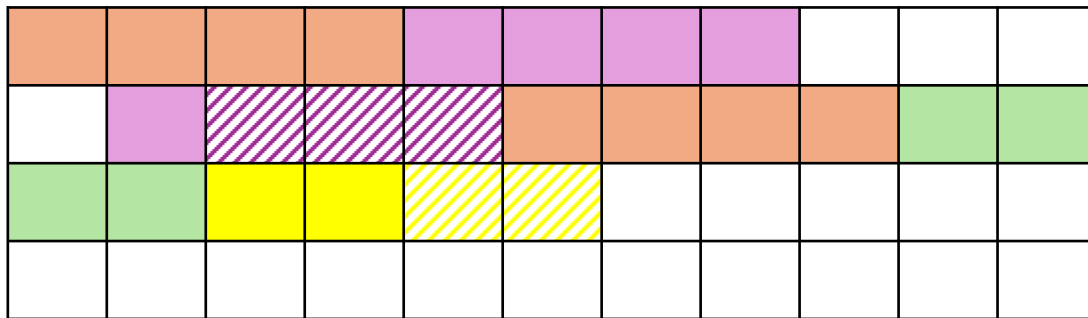




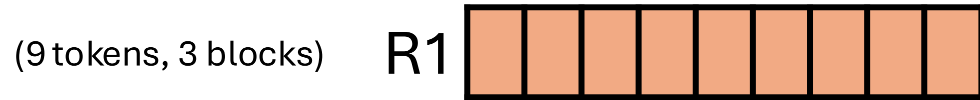
GPU Memory (block size = 4)

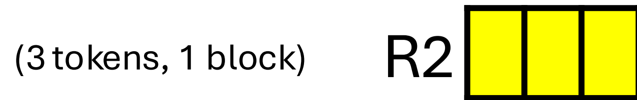
Memory Allocation with vLLM



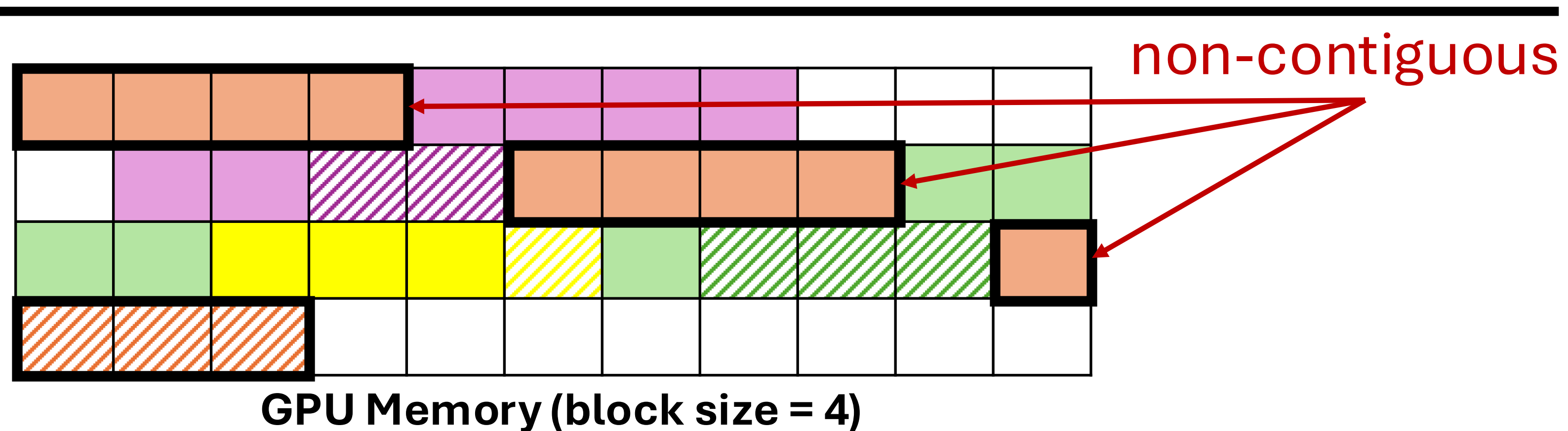


Memory allocation with vLLM





- Dynamic allocation eliminates fragmentation
- Makes KV-cache **non-contiguous**



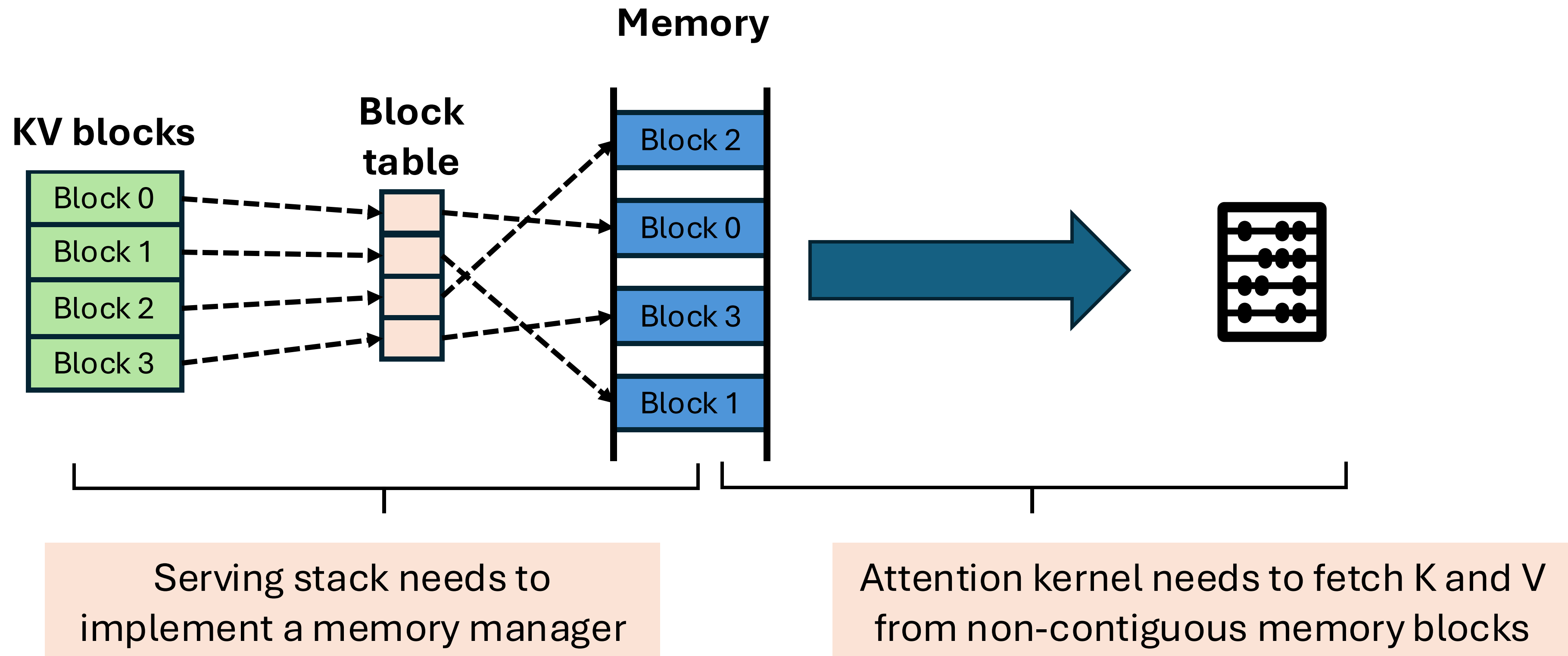
vLLM and Paged Attention

$$Attention(q_i, K, V) = softmax(\frac{q_i K^T}{scale})V$$

- Conventional implementations expect **contiguous K and V**
  - No longer possible in vLLM

- **PagedAttention**
  - Compute attention over non-contiguous blocks of K and V

Programming Overhead



Issues with PagedAttention

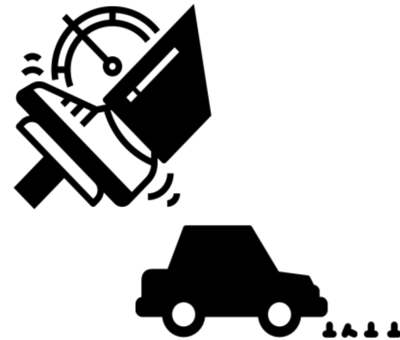
Programming

Writing performant GPU code is non-trivial

Issues with PagedAttention

Programming

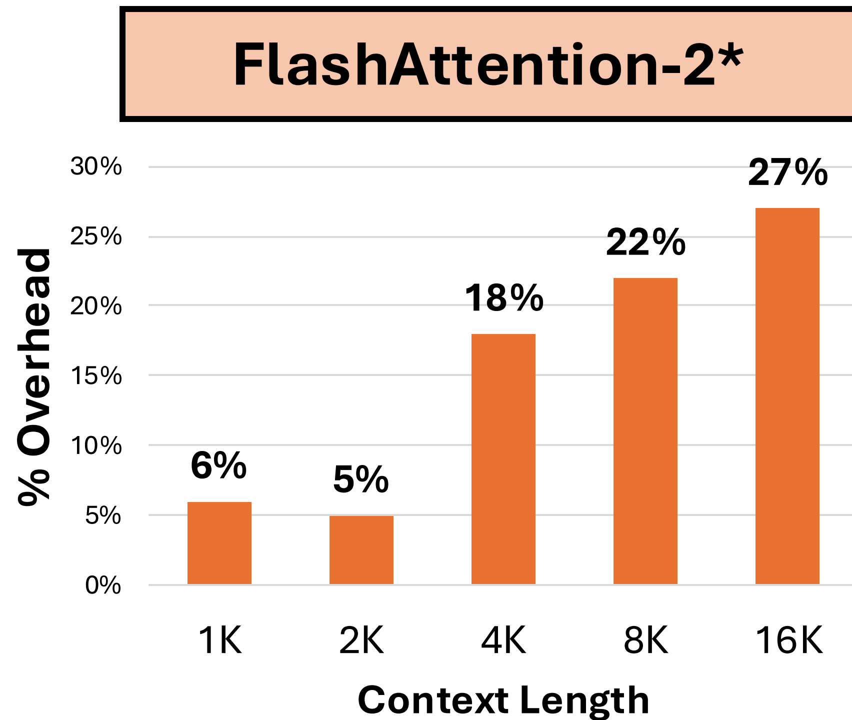
Writing performant GPU code is non-trivial

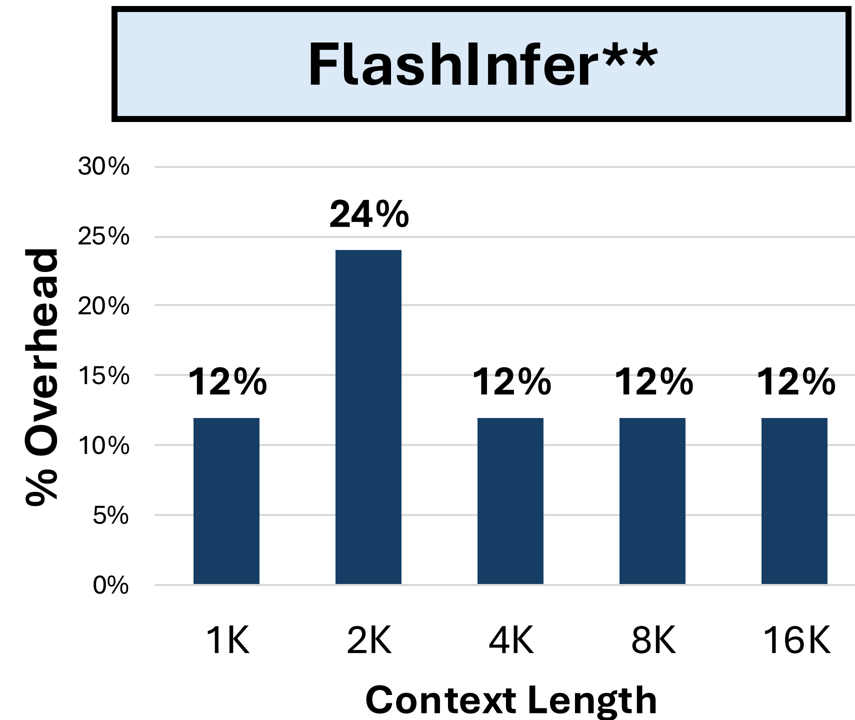


Performance

Redundant address translation has a cost

Performance Overhead



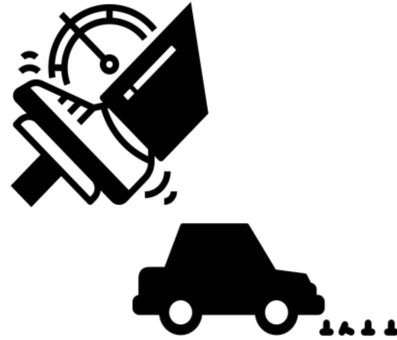


* Dao-AILab/flash-attention: Fast and memory-efficient exact attention (github.com)
** flashinfer-ai/flashinfer: FlashInfer: Kernel Library for LLM Serving (github.com)

Issues with PagedAttention

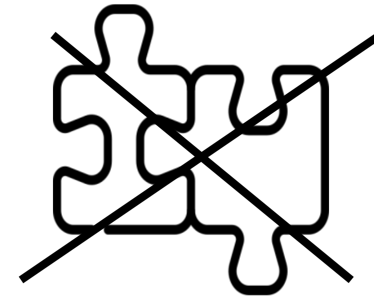
Programming

Writing performant GPU code is non-trivial



Performance

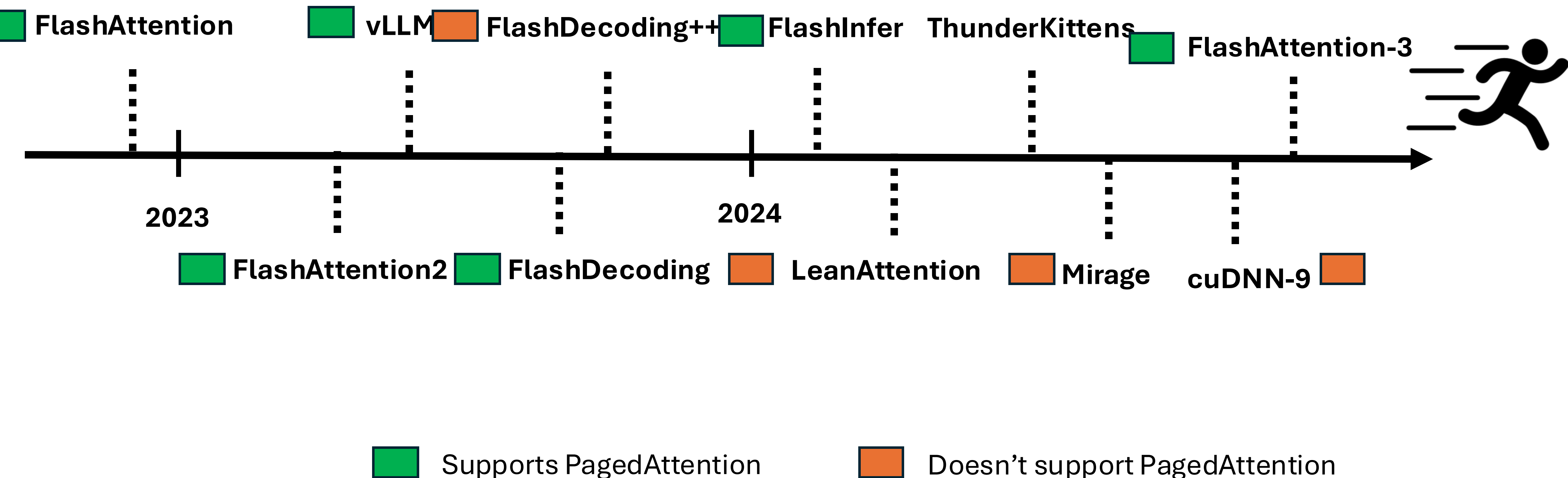
Redundant address translation has a cost



Portability

Kernels are not compatible!

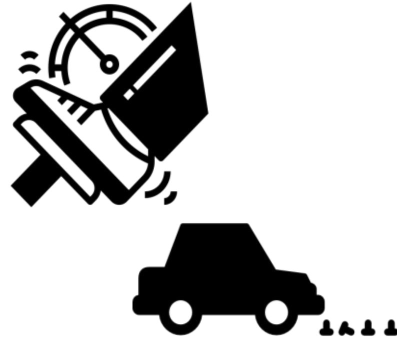
Why care about portability?



Issues with PagedAttention

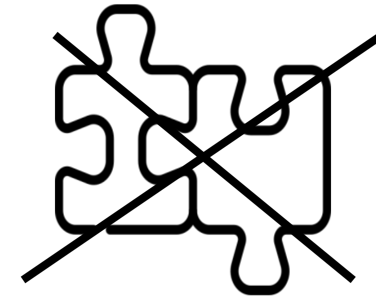
Programming

Writing performant GPU code is non-trivial



Performance

Redundant address translation has a cost



Portability

Kernels are not compatible (different formats)

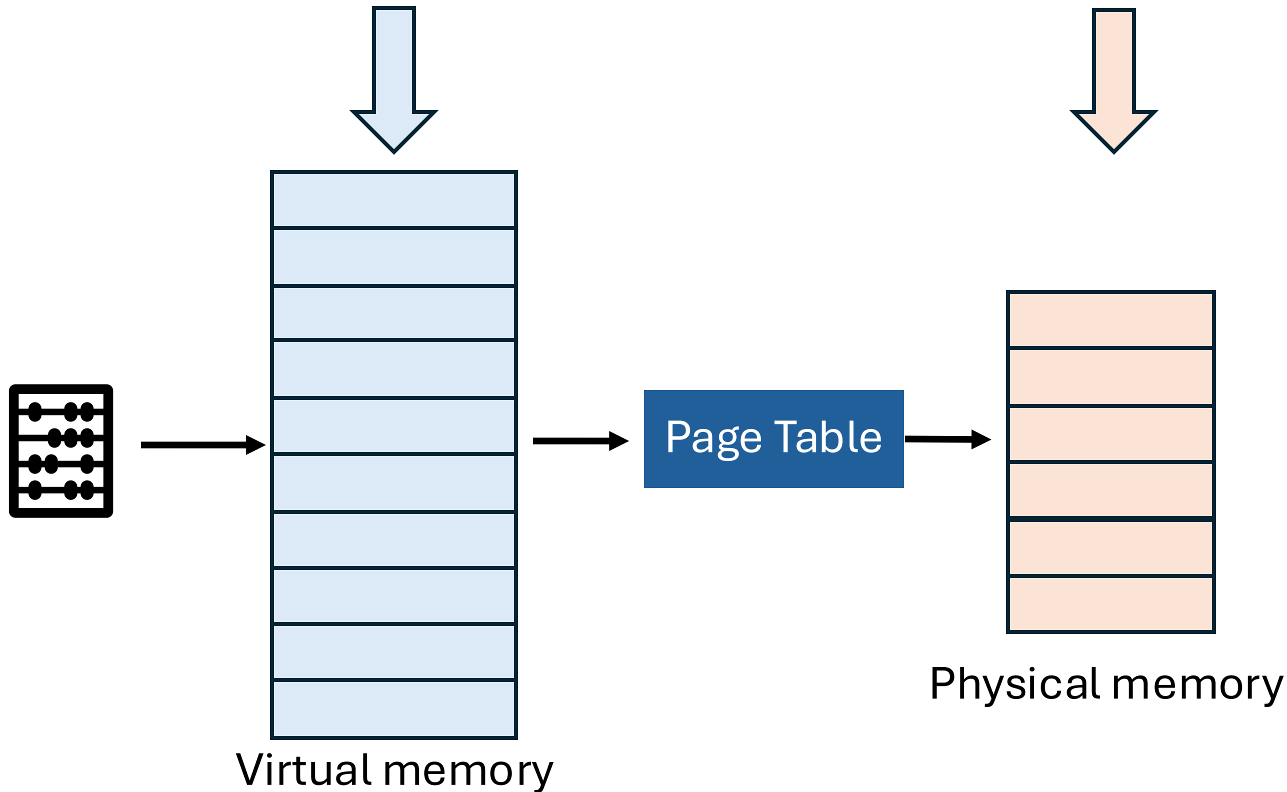
Can we do better?

- Non-contiguous layout is not ideal.
- Ideal solution:
  - **Dynamic** memory allocation
  - **Contiguous** memory layout

} These goals are usually conflicting

Can we resolve the conflict?

Enabling contiguous dynamic allocation



Virtual memory is abundant
(128TB per process)

allocate large chunks, ahead-of-time

Physical memory is limited
(80GB per GPU)

allocate small chunks, on demand

Key idea: Let's allocate them separately

vAttention

Decoupling virtual and physical memory allocation

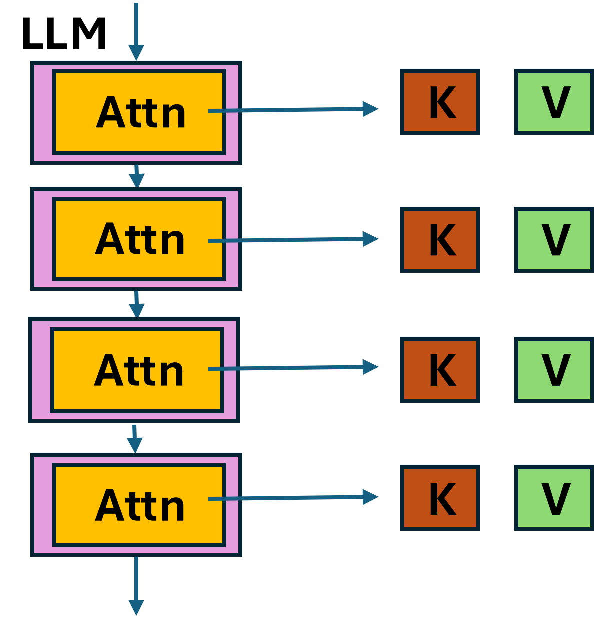
Leveraging system support for demand paging

Optimizations (co-design with LLM inference)

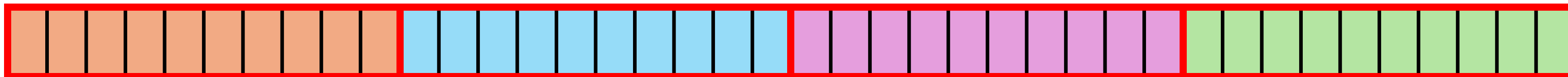
Memory Allocation in vAttention

Each worker allocates 2*N virtual buffers in advance

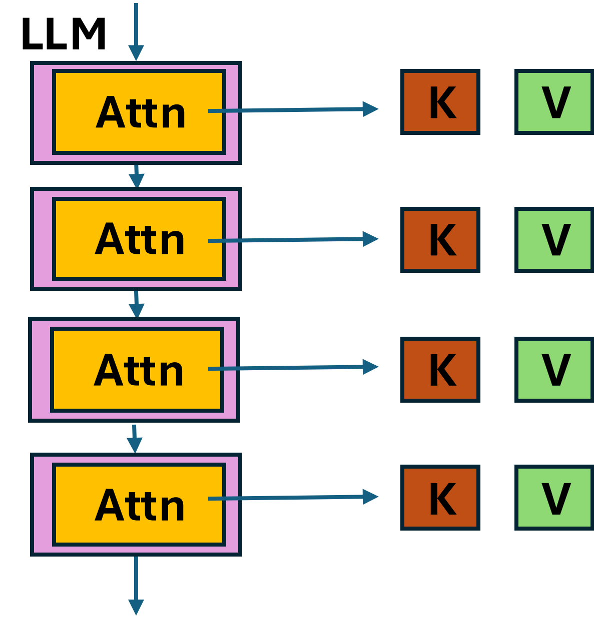
- N = number of layers hosted on the worker
- Separate tensors for K and V at each layer
- Buffer size based on max context length and batch size



One virtual buffer (batch size=4, each block is a token)



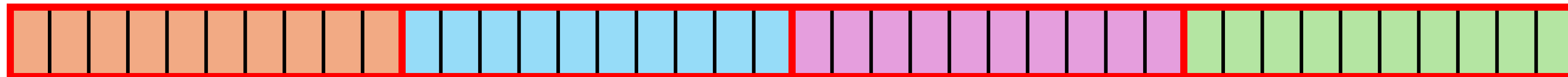
Memory Allocation in vAttention



Each worker allocates 2*N virtual buffers in advance

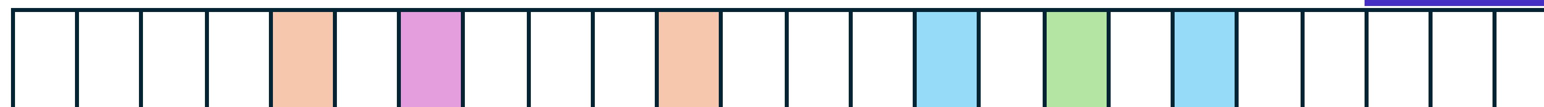
- N = number of layers hosted on the worker
- Separate tensors for K and V at each layer
- Buffer size based on max context length and batch size

One virtual buffer (batch size=4, each block is a token)



Page Table

Comment: How is the right batch size computed?



Physical Memory (each block is a page)

How is this feasible?

Create contiguous virtual memory!

CUDA VM APIs	vAttention VM APIs	Description	Latency (microseconds) 64KB	128KB	256KB	2MB
cuMemAddressReserve *	vMemReserve *	Allocate a buffer in virtual memory	18	17	16	2
cuMemCreate *	vMemCreate *	Allocate a handle in physical memory	1.7	2	2.1	29
cuMemMap	vMemMap	Map a physical handle to a virtual buffer	8	8.5	9	2
cuMemSetAccess	-	Enable access to a virtual buffer	-	-	-	38
cuMemUnmap	-	Unmap physical handle from a virtual buffer	-	-	-	34
cuMemRelease *	vMemRelease *	Free physical pages of a handle	2	3	4	23
cuMemAddressFree *	vMemFree *	Free a virtual memory buffer	35	35	35	1

CUDA APIs allow implementing vAttention

Create dynamic physical memory!

Comment: How are page faults handled?

Challenges for vAttention

Allocating a physical memory page requires a syscall

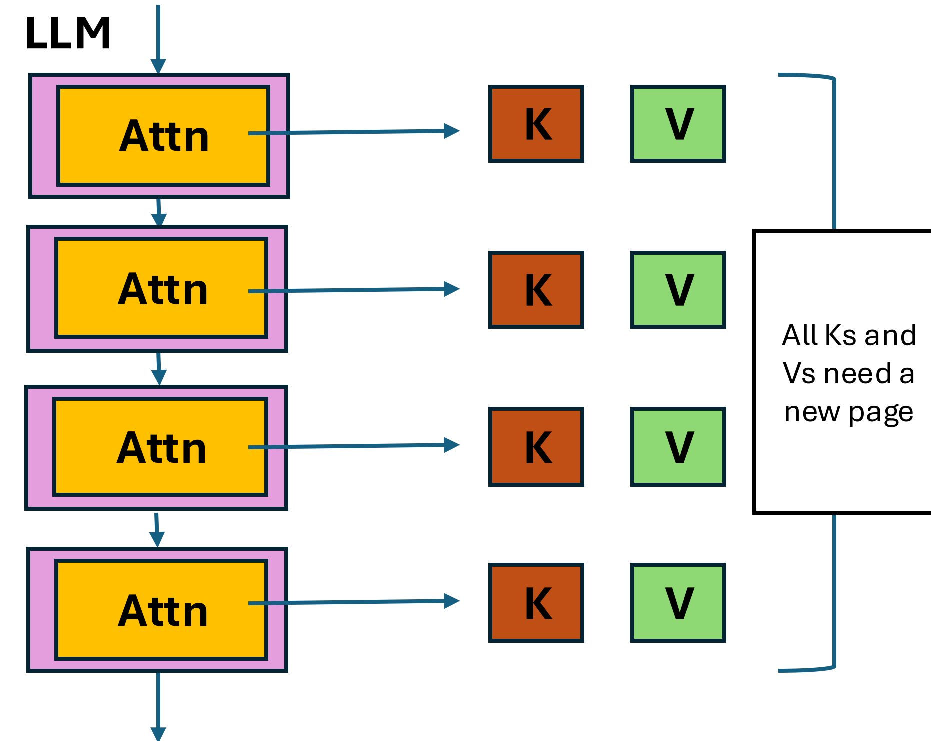
High Latency

CUDA drivers allocate only large pages (>=2MB)

High Fragmentation

Challenges 1 – Memory Allocation Latency

- CUDA API calls are expensive - Allocating one page takes **~40 μs**
- Need to allocate multiple pages at once
  - Latency overhead grows proportionally
- Example: **Yi-34B**
  - 60 layers == 120 virtual tensors
  - **5ms** (=120*40 μs) latency overhead per request
  - **50ms** overhead if 10 requests need new pages at once



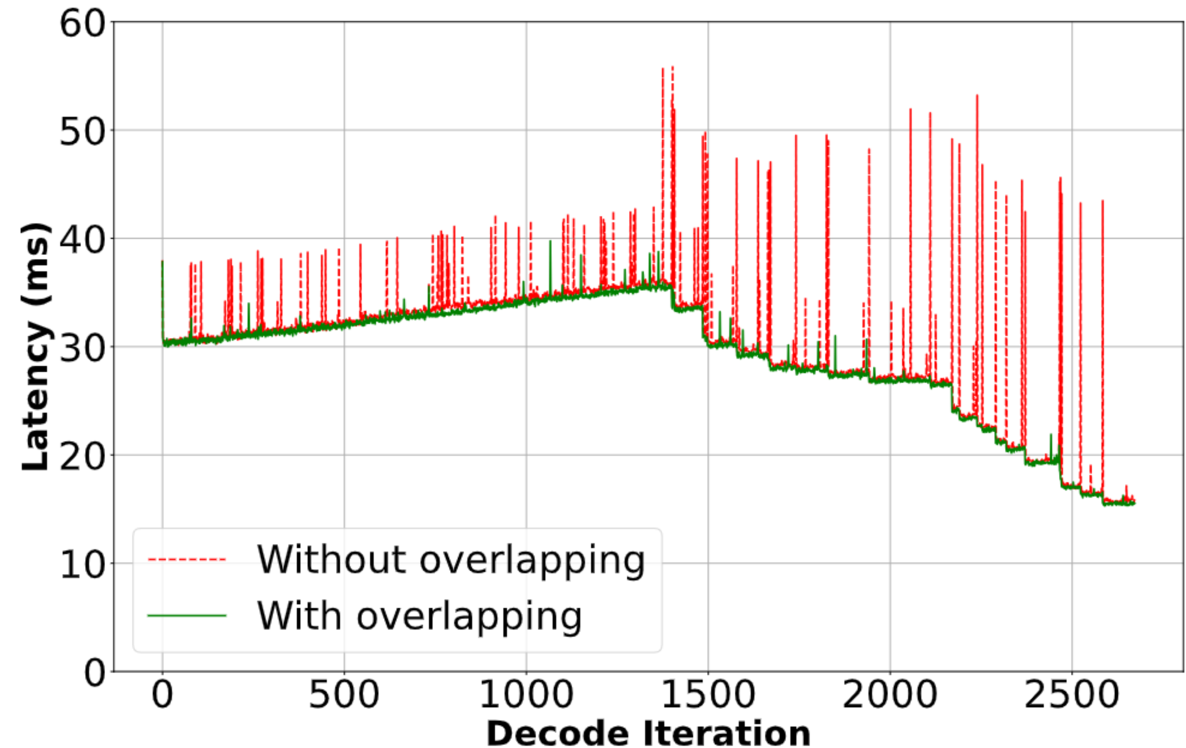
Optimization: Overlap allocation with compute

- Each decode iteration generates **one** token (per request)

- Memory requirement is known ahead-of-time (dream property!)

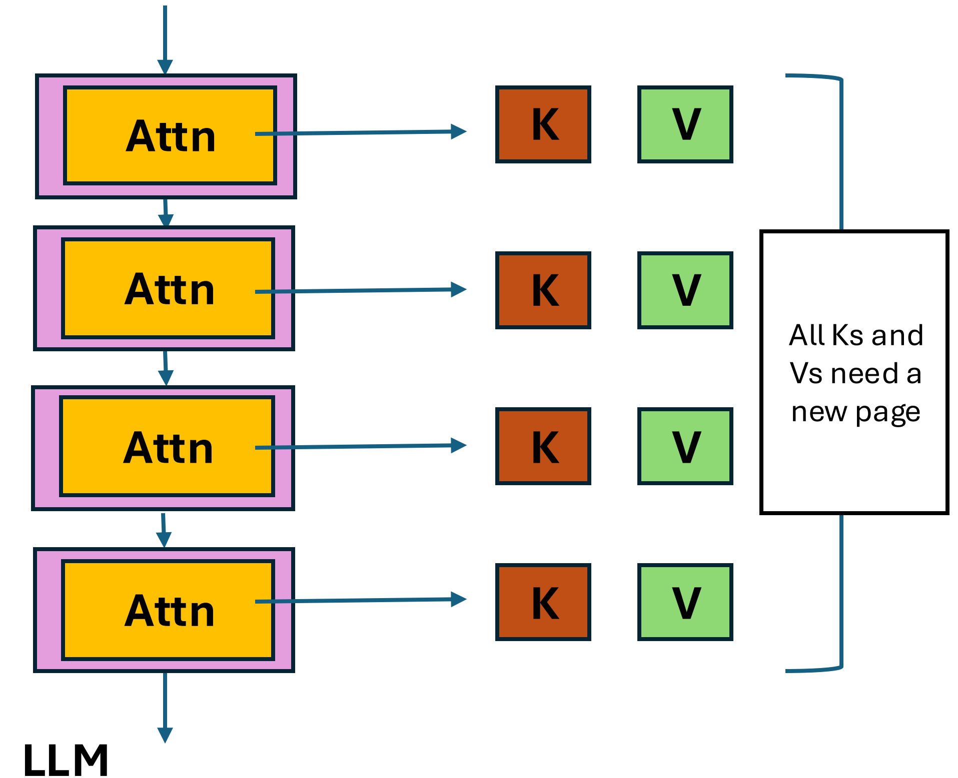
💡 Track progress of each request to determine when a new page is required

💡 **Asynchronously** allocate pages for iteration **i+1** when iteration **i** is executing



Challenge 2: Fragmentation (Physical memory)

- Min page size in CUDA is **2MB**
- vAttention allocates 2*N pages at once
- Fragmentation proportional to:
  - Number of layers
  - Degree of tensor-parallelism



Challenge 2: Fragmentation (Physical memory)

- Example: **Yi-34B**
  - 60 layers == 120 virtual tensors (per TP-worker)

Maximum memory wasted (per request) for Yi-34B

TP Dimension	Max Memory wasted
1	240MB
2	480MB
4	**960MB**
8	**1920MB**

Optimization: Allocate smaller physical pages

GPUs natively support 4KB, 64KB and 2MB pages.

- **Solution:** Update CUDA drivers to allocate small (64KB) pages
- **Challenge:** CUDA drivers are closed source, so all CUDA APIs had to be rewritten

Maximum memory wasted (per request) for Yi-34B

TP dimension	64KB	2MB
1	7.5MB	240MB
2	15MB	480MB
4	30MB	960MB
8	**60MB**	**1920MB**

Up to 96% reduction in memory wastage

Comment: This can further increase the overhead of memory allocation

vAttention Challenges

Allocating a physical memory pages requires a syscall

High Latency

Async allocation

CUDA drivers allocate only large pages (>=2MB)

High Fragmentation

Small pages

vAttention Optimization

Deferred reclamation

If a new request joins right after an old request has completed, transfer the physical and virtual memory.

Eager Allocation

Pre-allocate some virtual tensors even before a request arrives

Evaluation

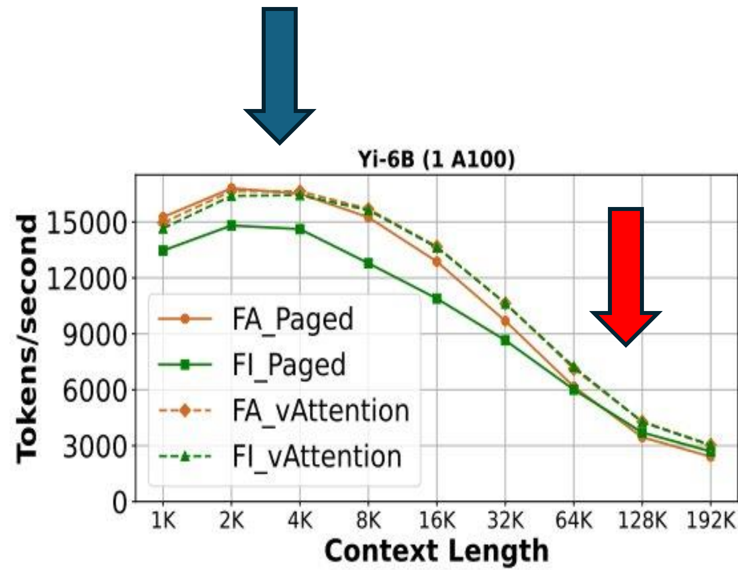
Model	Hardware	# Q Heads	# KV Heads	# Layers
Yi-6B	1 A100	32	4	32
Llama-3-8B	2 A100s	32	8	32
Yi-34B	2 A100s	56	8	60

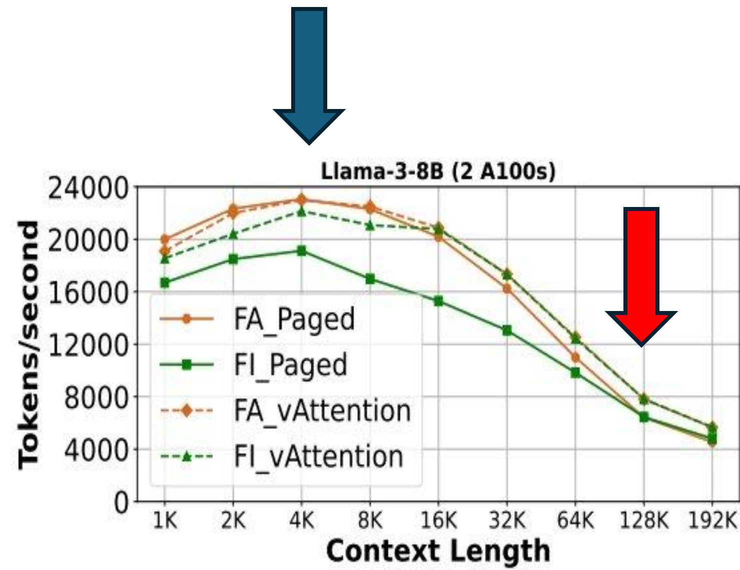
- Prefill and decode phases
- LLM serving throughput
- Effect of each of our optimizations

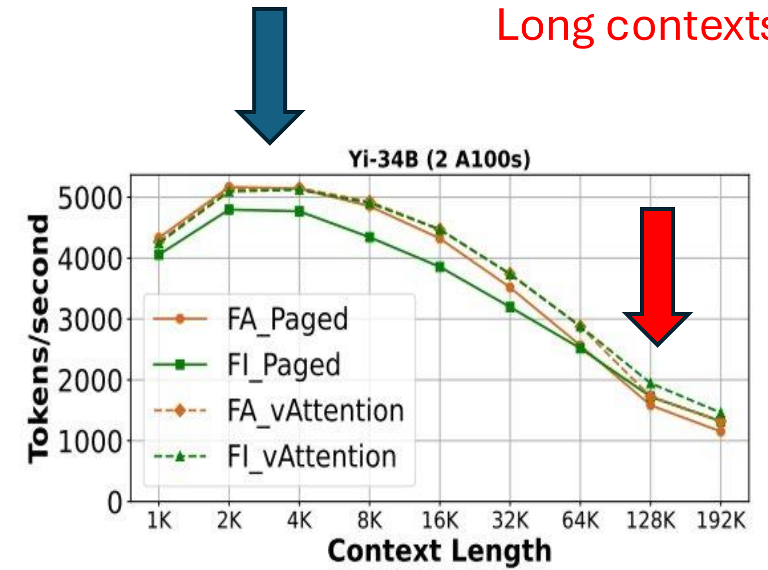
Prefill Evaluation

Small contexts

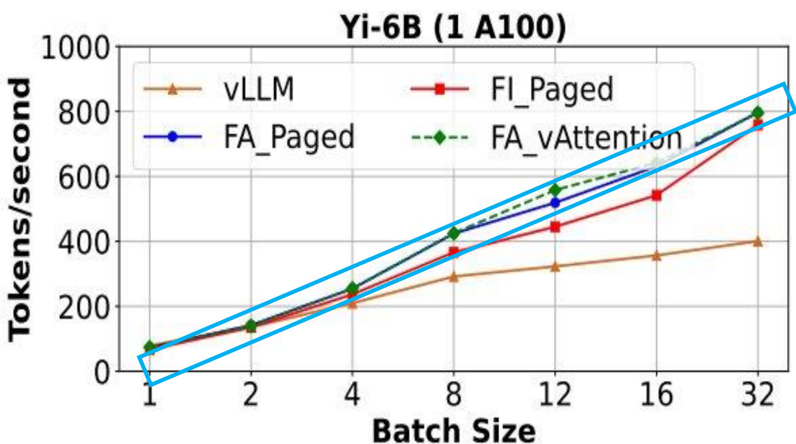
Long contexts

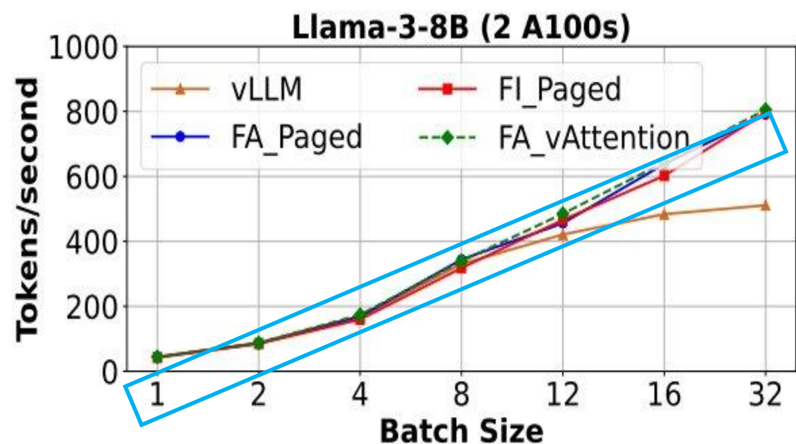


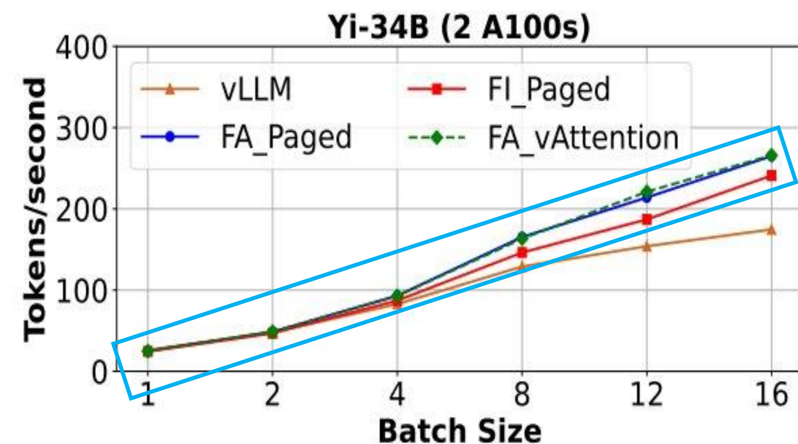




Decode Evaluation

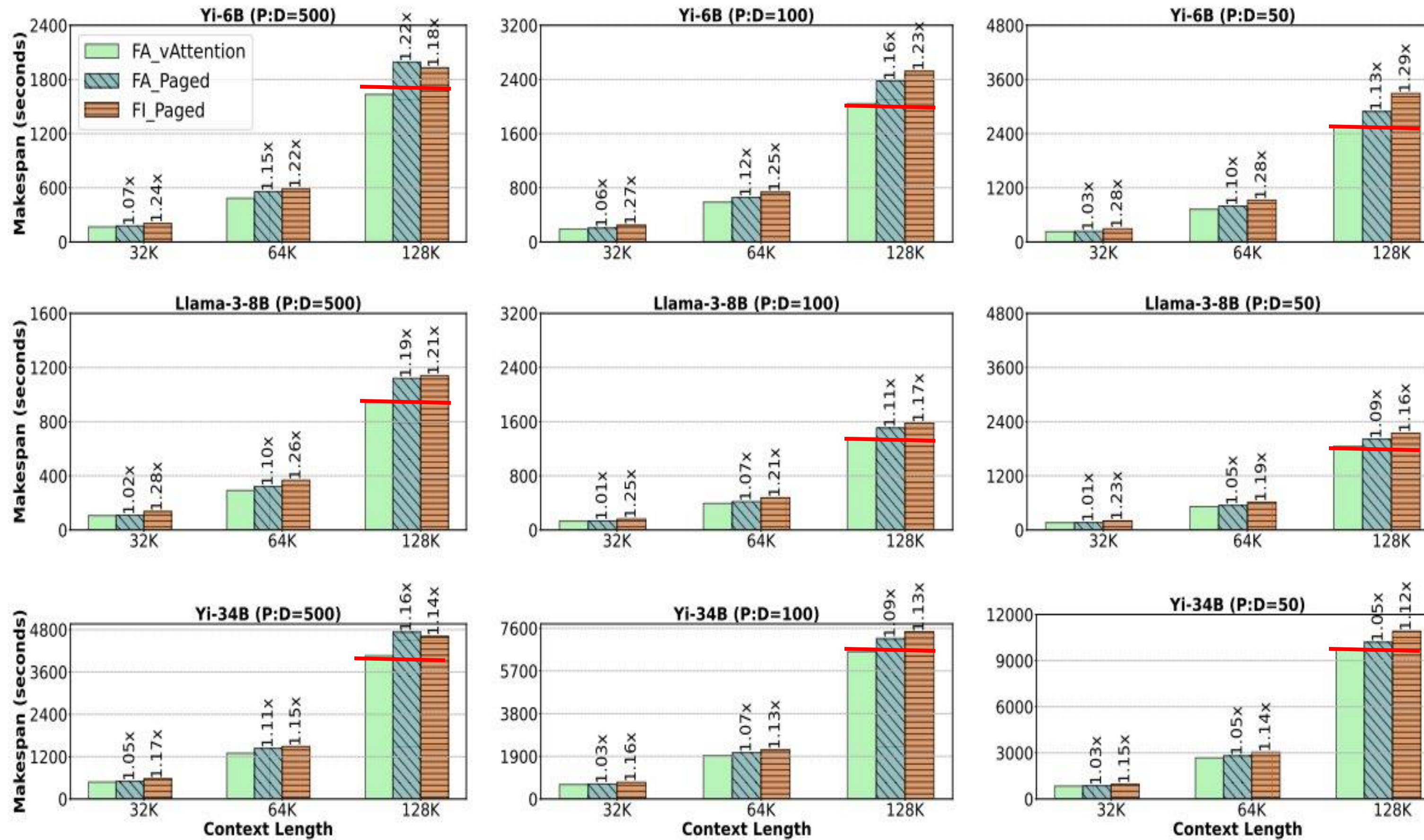






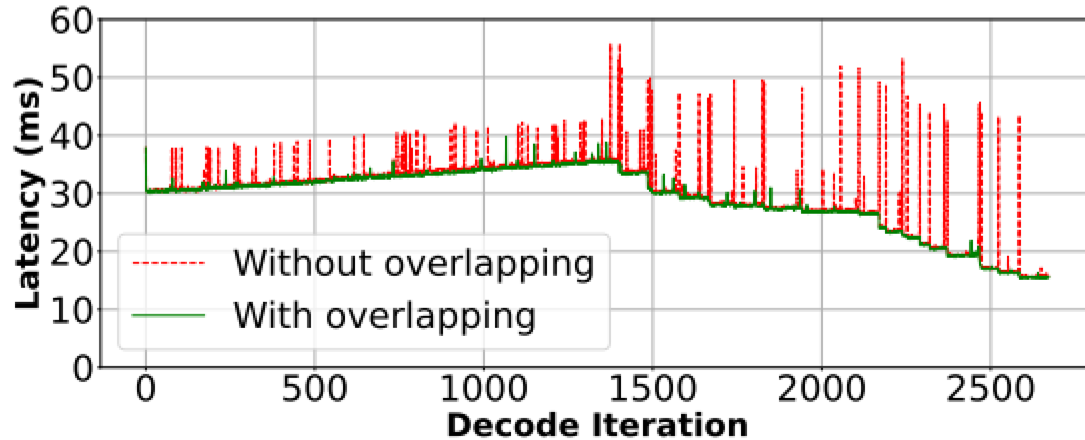
Model	BS	vLLM	FA_Paged	FI_Paged	FA_vAttention
Yi-6B	16	32.3	11.5	15.2	11.3
	32	64.1	25.5	25.4	25.3
Llama-3-8B	16	17.8	11.9	12.1	11.8
	32	35.3	25.4	23.23	25.3
Yi-34B	12	41.4	17.4	24.1	17.4
	16	55.1	21.7	28.8	21.8

End-to-end Throughput Evaluation



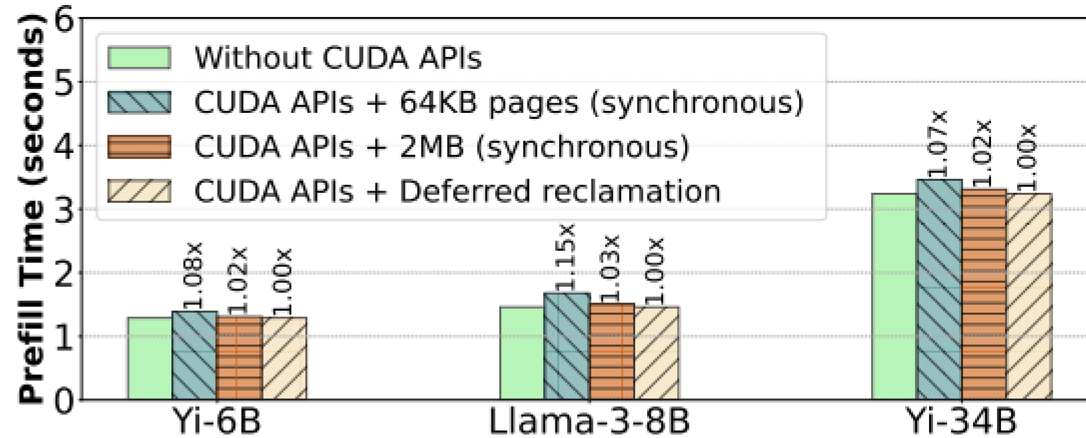
•P:D-ratio of prefill to decode tokens

Hiding allocation latency



This experiment used a batch of 32 requests and initialized the prefill context length of each request to be in between 4K–8K (chosen randomly).

Deferred reclamation

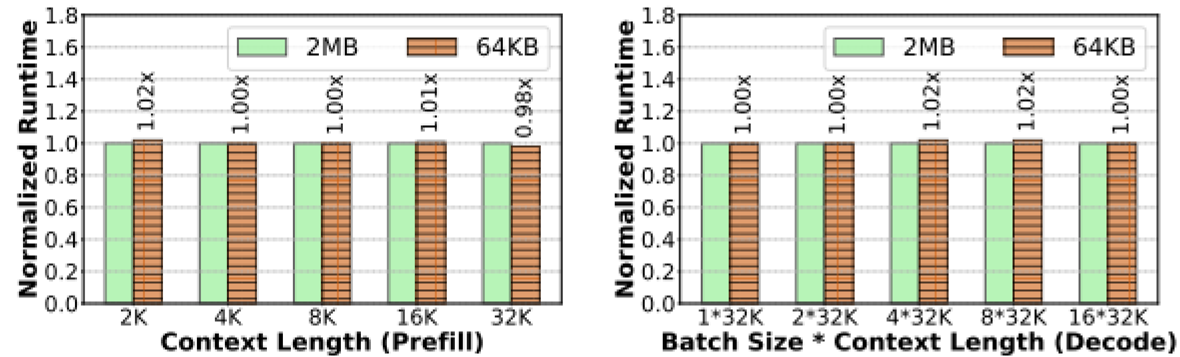


- Synchronous memory allocation using CUDA APIs for prefills incurs overhead of up to 1.15× with 64KB pages and up to 1.03× with 2MB pages.

- Memory allocation bandwidth

Config.	64KB	128KB	256KB	2MB
TP-1	7.59	14.56	27.04	35.17
TP-2	15.18	29.12	54.08	70.34

- Effect of page size



- Programming Effort

```
-import flash_attn as fa
+import flashinfer as fi

-def flash_attn_prefill(q, k_cache, v_cache, kv_len):
-    k = k_cache[:, :kv_len, :, :]
-    v = v_cache[:, :kv_len, :, :]
-    return fa.flash_attn_func(q, k, v, causal=True)
+def flash_infer_prefill(q, k_cache, v_cache, kv_len):
+    q = q.squeeze(0)
+    k = k_cache.squeeze(0)[:kv_len, :, :]
+    v = v_cache.squeeze(0)[:kv_len, :, :]
+    return fi.single_prefill_with_kv_cache(q, k, v, causal=True)
```

vAttention

- **vAttention:** An alternative to PagedAttention
  - Leveraging system support for demand paging

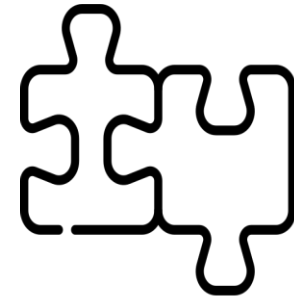
Strengths

Programming

Supports unmodified GPU implementations

Performance

Not impacted by dynamic memory allocation



Portability

Adopting new kernels is very easy

vAttention

- **vAttention:** An alternative to PagedAttention
  - Leveraging system support for demand paging

Room for Improvement

Adding support for non-NVIDIA architectures

Bringing up to parity with GMLake (Same approach as vAttention, but for training)