Agenda:

- Memory management in LLM Serving & Paged Attention
- Batching, Throughput, Latency
- motivation for continuous batching
- "Continuous" Batching in LLM systems: if time

* First: clarification on division of Q,K,V for flash attn

+ consider a single attn head, where for one xemple, Q,K,V are all $N \times d$

1) Divide Q into blocks of size $B_r \times d$
   ↳ $T_r = \frac{N}{B_r}$ blocks

2) Divide K,V into blocks of size $B_c \times d$
   ↳ $T_c = \frac{N}{B_c}$ blocks

↱ iterating over Q

→ For $1 \le i \le T_r$

- Load $Q_i$ ↱ each block output
- Initialize $O_i^{(0)}$, $\ell_i^{(0)}$ ↱ each log sum exp, $m_i^{(0)} = -\infty$ → max

→ For $1 \le j \le T_c$ → iterating over K,V

- calculate local numerator, denom
- using previous output, calculate this output (iterative)

> where warp scheduling comes into play.

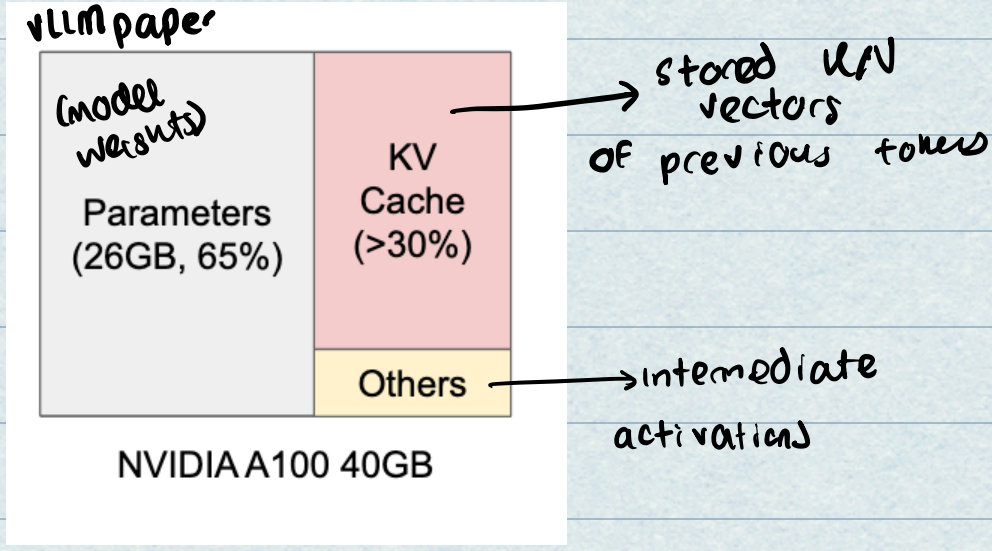
→ split attn heads, Q over thread blocks to fill S.M.s

→ what about warps?
↳ within a block, FA1 splits K,V across warps, Q accessible to all warps
↳ instead, FA2: split Q across warps, K,V accessible by all
↳ think about inner inner part of For loop → we have specific K,V block, Q block → how do we warp schedule

*see FA2 paper for more details!

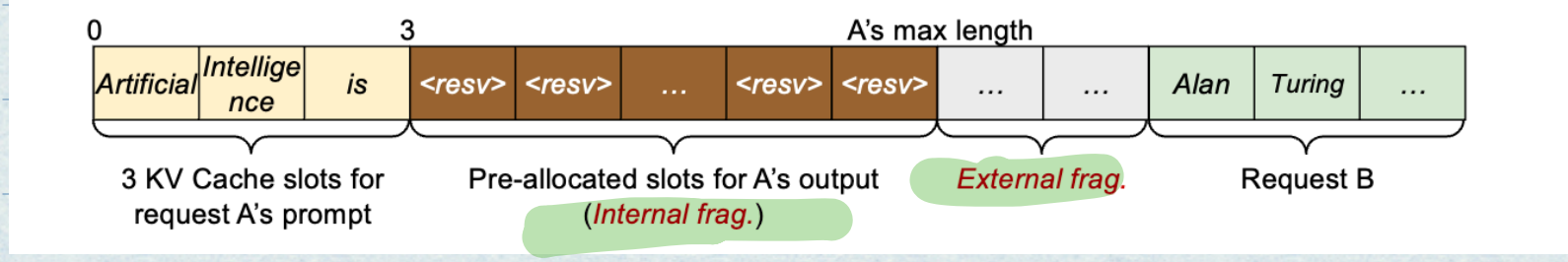
# FIRST PART OF LECTURE: PAGED ATTENTION

- vLLM's original main technique
- core question: how do we MANAGE GPU memory (for KV cache, specifically)




- a few things make memory management hard:
  1. For each request doing autoregressive decoding, we DON'T KNOW how long the generation will be
  2. Across requests, users can provide dif. max request lengths

* previous engines allocated contiguous spaces of memory in KV cache for each request's max memory:



* some observations:

- A and B have Dif max request lengths
- engine pre-allocates slots up to max length

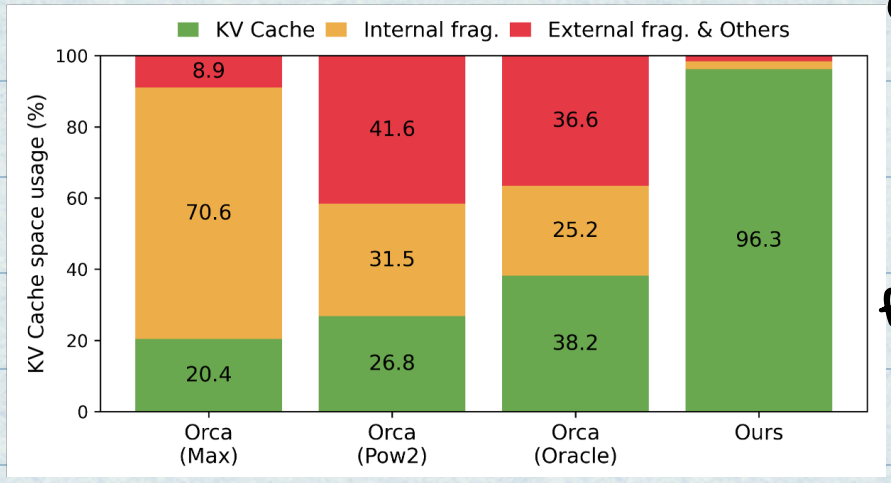
- this "static memory management":
  - leads to 2 types of fragmentation

1) Internal frag: LLM does not predict up to max request length. wasted space

2) External frag: when allocating memory requests of dif. sizes, there will always be some unused space

*go study memory allocation schemes for more info

FRAGMENTATION IS SIGNIFICANT:



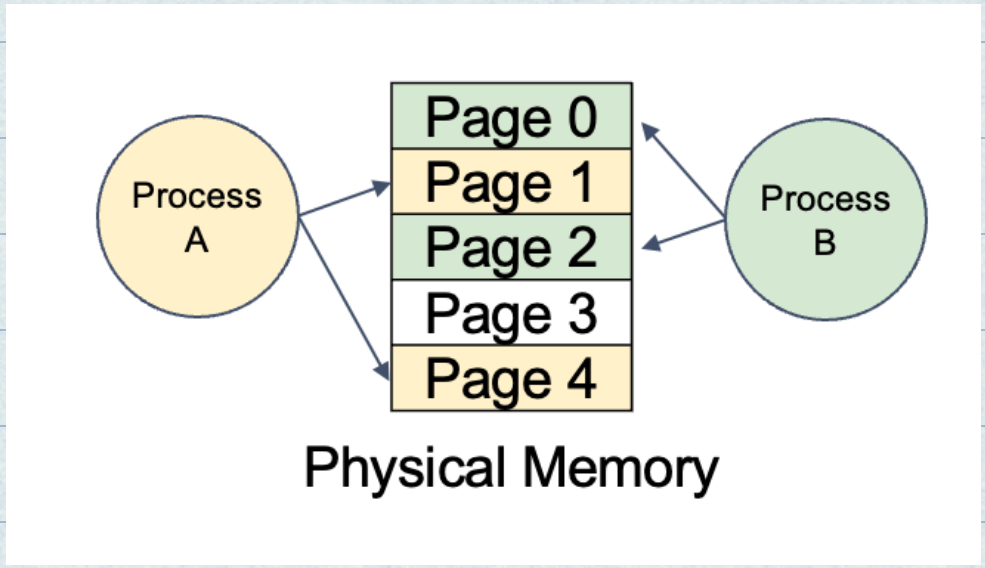

(from vLLM paper):
- in previous baselines - only 20-40% of available memory for KV cache is used for useful data

- GPU device memory is the bottleneck - we really shouldn't be wasting any!

KEY INSIGHT OF VLLM WORK:

- we've already solved a similar problem w/ OS-level paging and virtual memory:

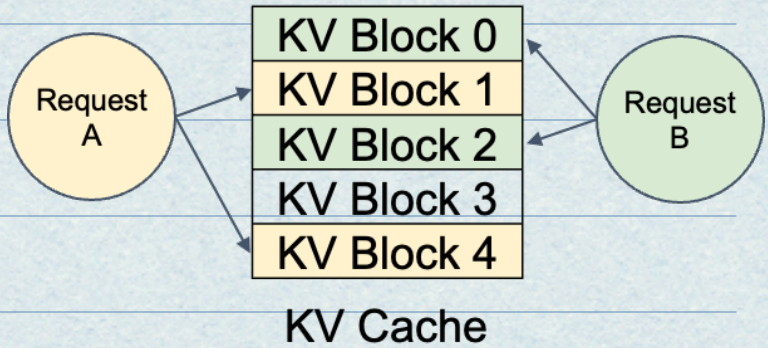
  - OS divides up memory into "pages"
  - will map program LOGICAL pages to PHYSICAL pages
    - also logical pages need not be physically contiguous:




But each process THINKS it has logically contiguous memory

Let's do something similar for the KV cache!

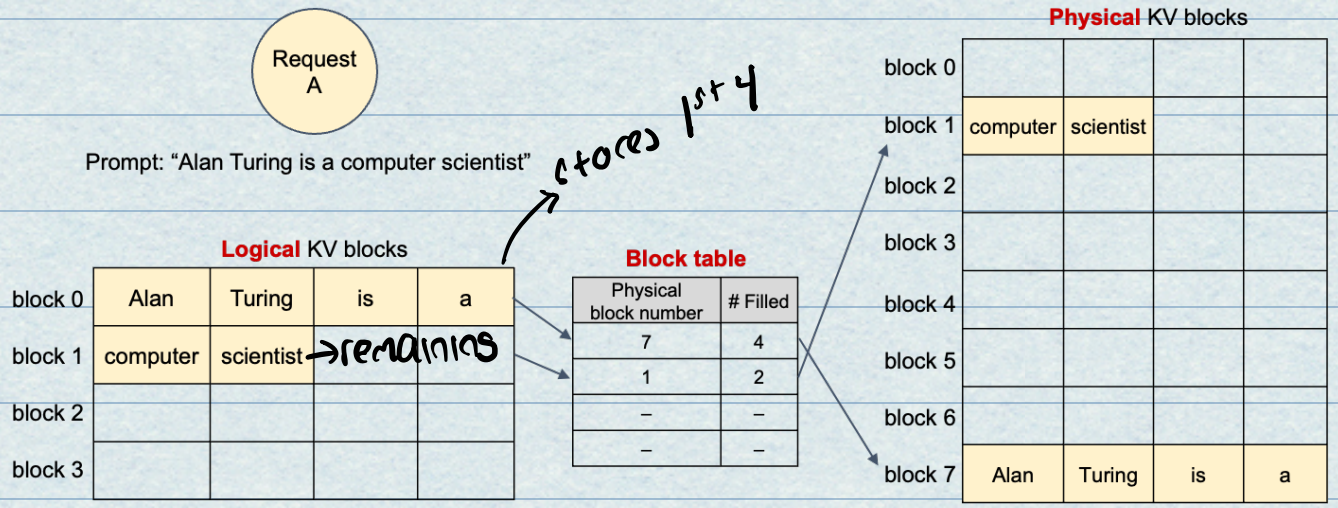
- partition KV cache into "blocks"
- to minimize fragmentation, blocks for dif. requests do NOT have to live next to each other:



KV-block = fixed-size contiguous memory chunk that stores KV states from left to right.

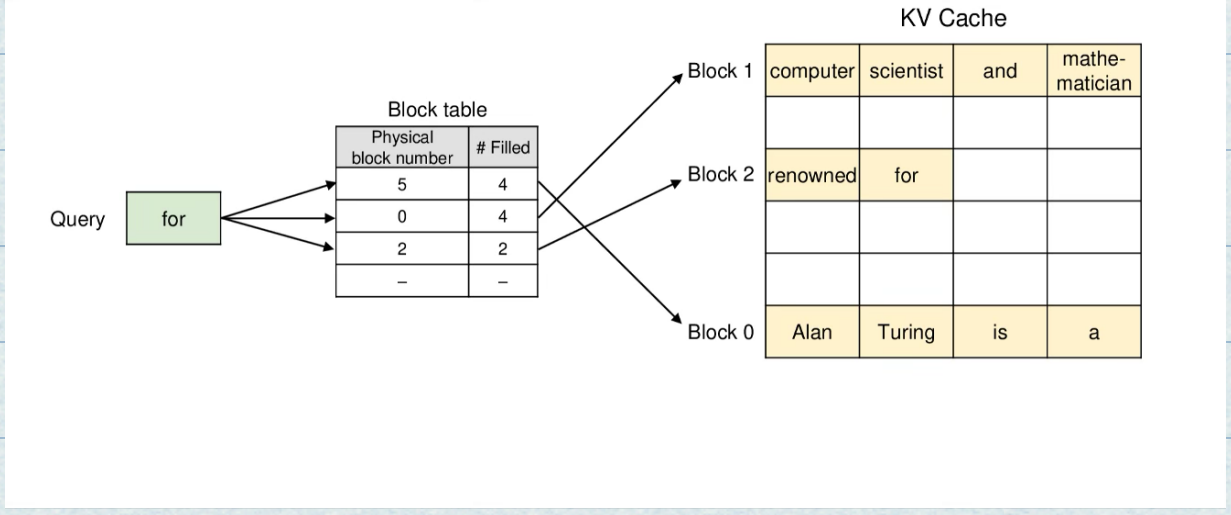
↳ block-size of 4 would be able to store k/v projections for 4 tokens

*now we can VIRTUALIZE THE KV CACHE:




↳request "thinks" B0 and B1 are contiguous, but they are physically not: (w/ block/page table in between)

*HOW do we compute attention?



- we need to compute attention for query "for" - w/ respect to each token before, but KV blocks cannot be fetched all at once!

- iterate over blocks containing KV cache for this request:
  - for each logical block, set row of block table
  - look up physical block location
  - calculate attn of query w.r.t those keys/values:
    - we are trying to calculate an attn SCORE for each token so far:
    - consider block j and we are trying to compute attn score of token i (i is query)

→ contains keys/values for token index:

$B \cdot (j-1)+1 \dots B \cdot j$

↳ for B=4;

$K_1$ → keys for tokens 1...4

$K_2$ → keys for token 5...8

$V_1$ → vals for token 1..4

$V_2$ → vals for token 5-8

$1 = 4(1-1)+1$

$5 = 4(2-1)+1$

$$A_{ij} = \frac{e^{\frac{q_i \cdot K_j^T}{\sqrt{D}}}}{\sum_{t=1}^{i/B} e^{(q_i \cdot K_t^T / \sqrt{D})}}$$

($q_i$: $1 \times D$, $K_j^T$: $D \times B$)

$A_{ij}$ → attn scores of token i, block j

$$O_i = \sum_{j=1}^{i/B} V_j A_{ij}^T$$

→ iteration for t=1.. i/B → all blocks so far! → gets w proper softmax sum

→ in $O_i$: again iterate from j=1 to i/B to get all values

→ one more note: how do we make space for K/V of newly predicted tokens?

↳ first: try to fill end of existing block

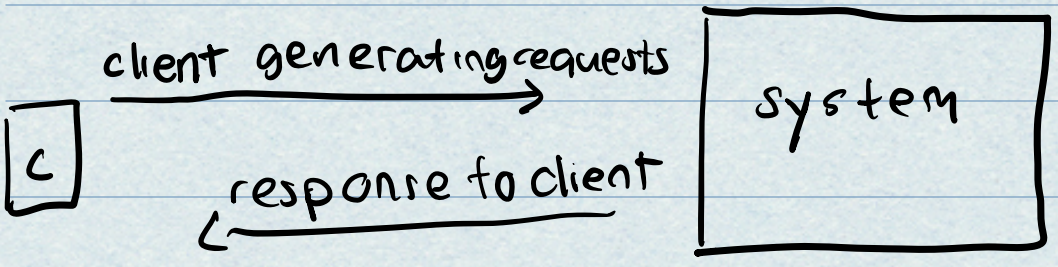
↳ then: allocate new blocks

*these kind of techniques open way for:

- prompt K/V caching ACCOSS requests
- prompt "tree" caching (e.g., where requests share some lowest common prefix

...and more!

## PART 2 OF LECTURE: BATCHING, THROUGHPUT, LATENCY




* latency: how long does it take for the system to process a request? measurement: time (e.g., sec or ms)

↳ generally - its easier to measure 2-way latency (there, processing, and back)

- in some cases - the time btwn C

and S is negligible - so we are effectively measuring processing latency

* throughput: how many bytes are being transferred over the network in unit time?
  ↳ measured in Data/s e.g. bytes/s
- throughput could include "not-useful" bytes (e.g., overheads like packet headers)
- therefore, another useful measure is:

*goodput - how much USEFUL DATA is being sent over network
  ↳ could be measured in something like "requests/s"

*utilization: how much of the available processing capacity are we using?
  ↳ we saw this w/ memory vs. compute utilization

* Discuss for 5 minutes: why can achieving BOTH low-latency (across all users) and high goodput be conflicting goals?

- we can achieve low-latency at expense of goodput or throughput:

- always admit 1 request to be processed at a time

- drop all requests that arrive while the 1 request is being processed

- the 1 requests gets all of systems resources

- we can achieve high throughput at expense of latency:

  - wait for B requests to arrive

  - execute B at some time! usually beneficial if some amortization (e.g. load some model weights for B items)

    ↳ B x N x d → dimension

    batch size, LLM seq. length

  - but for requests at beginning of B:

    we've waited to execute them

* How do we think about/measure throughput and latency?

  * plug for CS 1675 (being offered in fall)

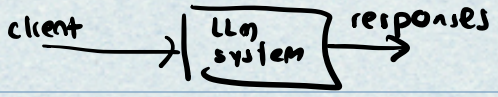
- lets first ignore batching → assume we get NO amortization from executing requests at some time:


Discussion

- why is lower right of graph better?

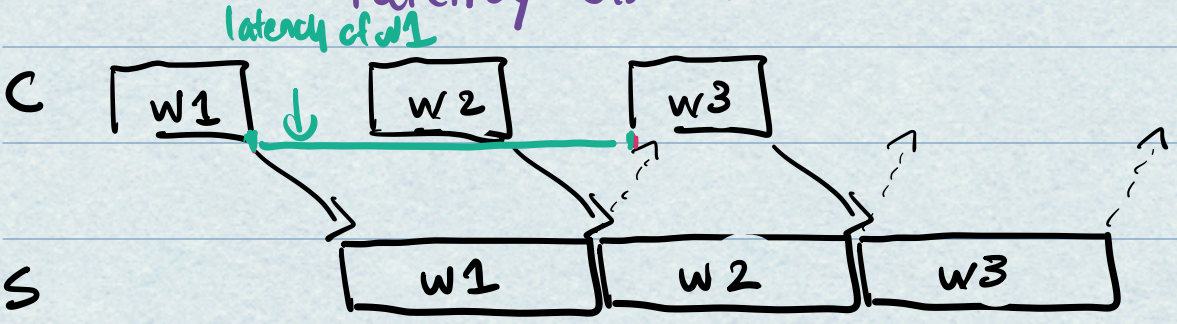
throughput/goodput
(measured)

- how would we measure thru?




* input: offered load
  ↳ client sends at some rate r

* output: achieved load
  latency distribution




- w2 − w1: 1/(client rate) → even:
(or w3 − w2)

- blue line: latecy of w1 → if we measure all latencies, we can get a distribution

- here: we are assuming client sending at even rate, all requests take same amount of time, system can't operate on more than 1 request per time

- let's break those assumptions!

1) in real world, clients may NOT be sending evenly at some # of requests per second (requests evenly spaced out over second)

↳ usually systems will have some kind of load balancer in front and enough replicas will be provisioned to handle load

→ take cs1675 to find how to model this in system measurement!

2) requests may NOT take same amt of time

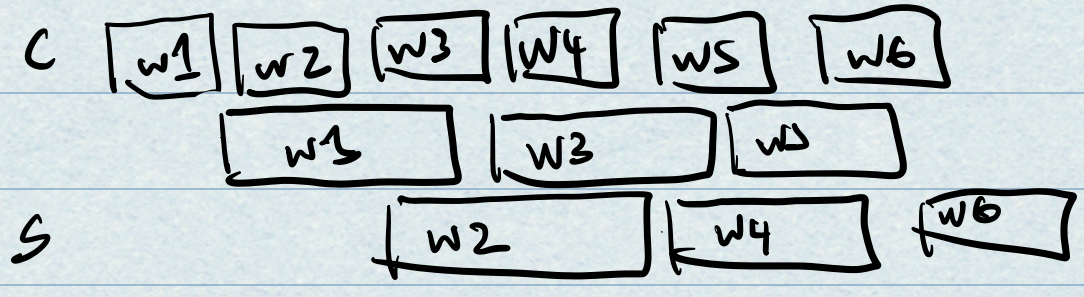
→ consider LLMs.

→ req 1 may generate 50 tokens, req 2 may generate 2 tokens

* this is one reason why generative AI systems are different!

· for CNN, all images take same amt of time to process!

3) our system may have capacity to execute multiple reqs at same time



- how might the amt of inference batching be controlled on a GPU?

- consider available memory to store KV cache!

- we have 7b param model, 40 GB GPU

→ 24 layers

→ D = 2048

→ seq. length (for all examples) = 1024

1) memory to load model:

$7 \times 10^{12} \cdot 2_{fp16} = 14\text{GB}$

2) remaining memory for KV cache: ~26GB

per token overhead for KV cache:

$2_{(K,V)} \times 2_{(fp16)} \times n_{layers} \times d_{model}$

$= 4 \times 24 \times 2048$

$= 200\text{k bytes} = .0002 \text{ GB}$

4) How many tokens can we fit in KV cache?

$$\frac{26 \text{ GB}}{.0002 \text{ GB}} \approx 130\text{K TOKENS}$$

5) for fixed seq. length, how many batch items can we fit?

$$\frac{130\text{K}}{1024} = 128$$

· naive algorithm: (at LLM GPU)

1) keep model weights loaded.

2) Wait for 128 inference requests.

3) Compute KV cache for all 128 requests and their prompts (tokens so far).

- But why might this algorithm be bad?

1: prioritizing throughput over latency

2: not all inference requests length 1024

*next time: continuous batching / scheduling