Agenda:

- Batching scheme we left off w/ last time
- Iteration-Level scheduling ("Continuous Batching")
- Different Inference Perf metrics: TTFT, TBT
- scheduling prioritization options (metric-dependent)
  - ↳ Decode vs. Prefill priortizing
- chunked prefill

* Resources:

→ Anyscale BlogPost: https://www.anyscale.com/blog/continuous-batching-llm-inference

→ ORCA OSDI Paper: https://www.usenix.org/conference/osdi22/presentation/yu

→ Saratni Serve Paper https://www.usenix.org/system/files/osdi24-agrawal.pdf

→ vLLM paper: https://arxiv.org/pdf/2309.06180

BATCHING SCHEME WE LEFT OFF WITH.

- Assumption: we are serving requests for a <u>SINGLE</u> model
- 7b model in fp16 takes up:

  $7 \times 10^{12} \cdot 2 \approx 14$ GB

- Approx. remaining of GPU memory (40 GB) can be used by KV cache

- LLM serving is quite memory bound (want to keep model weights loaded) - and serve multiple (a BATCH of requests)

* note - we still don't want to waste compute and recompute KV cache on each decode, so:

- when we serve a request, we want to make sure we can fit its entire KV cache in GPU memory

↳ otherwise recompute would be too much

- maximum batch size we can fit is therefore determined by:

1. overhead per token in KV cache
2. max sequence length any request can have

↳ $\underset{(K,V)}{2} \times 2 \times \underbrace{n_{layers} \times d_{model}}_{\substack{\text{in ex. from last class} \\ 24 \quad\quad 1 \quad\quad 2048}}$

$= .0002$ GB

$$\# \text{ tokens that can fit} = \frac{\overset{\text{mem. for KV cache}}{26\text{GB}}}{.0002\text{GB}} = 130\text{K}$$

max batchsize = 130K / $\underset{}{\overset{(1024)}{\text{max seq length}}}$ = 128

- "static" batching scheme we touched on:

1. wait until <batch size> requests are admitted to the system
2. For all batches, run prompt-processing step * to produce 1^{st} token *

3. Proceed one "iteration" at a time, where we run 1 decode step across all our examples in the batch to predict the <u>next</u> token in each sequence.

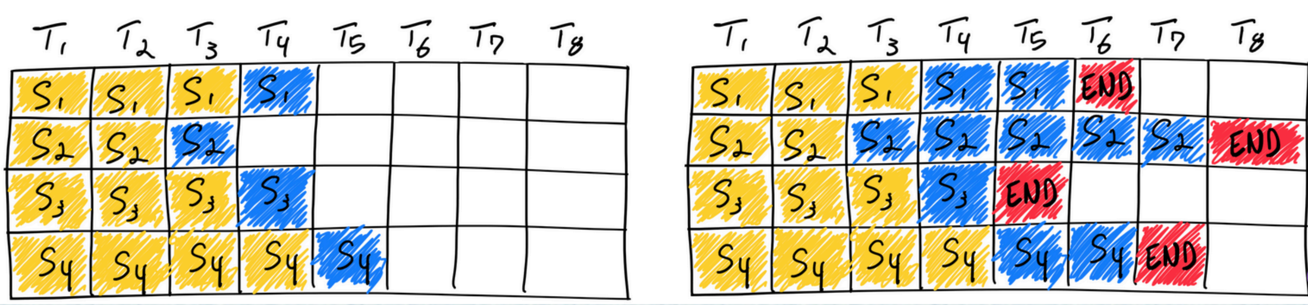
↳ if at any point, a sequence ends, don't do anything w/ that example until *all batch size* examples finish

* clarification: w/o paged attn memory management scheme from last time, "effective" batch size is lower (due to fragmentation, not all memory is available)

- say we have determined some batch size we know will fit, why is above scheme suboptimal?

*not all requests will use max # of tokens!!

https://www.anyscale.com/blog/continuous-batching-llm-inference



white boxes = idle gpu cycles

*how often is this a problem (likelihood of white squares?)

↳ depends on whether input sequences can together have *similar/same sizes*

↳ if all reqs produce 1 token (e.g., classification)

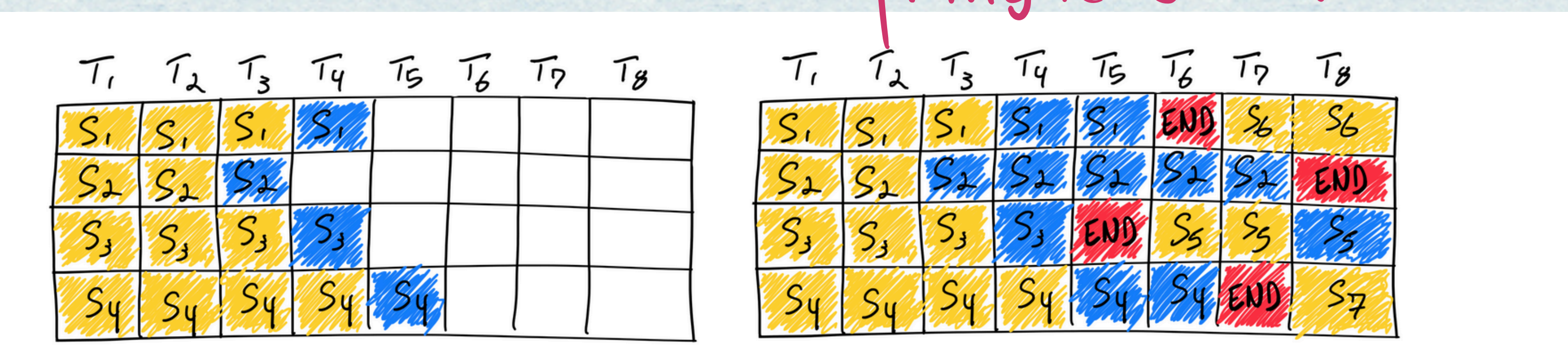
↳ if all prompts are same size

- in workloads like chatbots, there is a HIGH likelihood that sequences in same batch have dif. sizes

ITERATION-LEVEL SCHEDULING / CONTINUOUS BATCHING

- introduced by orca (OSDI 2022) - vllm implements this now

→ higher GPU utilization!



*handles late-arriving / early finish reqs more efficiently.

*key idea: let new requests start immediately if there is space

↳ at each iteration - check if a seq. has ended and whether to admit a new request

INFERENCE PERF METRICS

* TTFT – time to 1st token → how long does prefill take

* TBT – time between tokens → how long does it take to generate each subsequent token?

– there are 2 parts to serving (token prediction tasks):

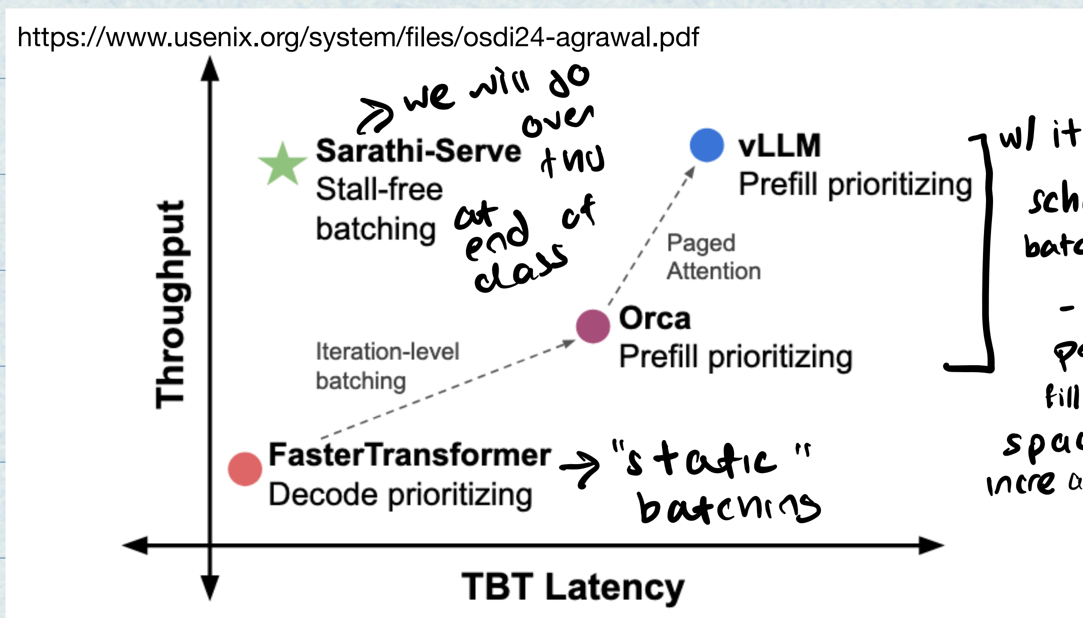
→ can be run in parallel (in one step)

1. Prefill: prompt processing to get 1st token
2. Decode: each subsequent iteration after

* prioritizing prefill will get you lower TTFT, higher throughput (first more before)

* prioritizing ongoing decodes will get you lower TBT, also will compromise throughput

* what does it mean to prioritize? Do these things come at a cost / is there a fundamental tradeoff?



(time for decodes)

w/ iteration-level scheduling – (continuous batching), we are:

– prioritizing prefill lets you fill in the empty space in batch, increasing throughput *why does prefill prioritizing sacrifice TBT / decode latency?

*let's look more closely at "static" batching and "continuous batching" scheduling algorithm:

· STATIC request-level BATCHING: optimizes TBT, but wastes GPU utilization (FasterTransformer)

https://www.usenix.org/system/files/osdi24-agrawal.pdf

```
Initialize current batch B ← ∅
while True do
  if B = ∅ then  → only when batch is empty, do we admit a new request
    R_new ← get_next_request()
    while can_allocate_request(R_new) do
      B ← B + R_new   ↳ if it "fits" in batch
      R_new ← get_next_request()
    prefill(B)
  else  → if B has ANYTHING at all, finish ongoing decodes
    decode(B)
  B ← filter_finished_requests(B)  → check if B is empty
```

→ DECODE PRIORITIZING = only admits new reqs if current decodes are all done

As we saw before, leads to low tput!

· what abt continuous batching?

```
Initialize current batch B ← ∅
while True do  → at each iter
  B_new ← ∅  → initialize set of new reqs to add
  R_new ← get_next_request()
  while can_allocate_request(R_new) do
      ↳ checks there is enough space for prefill of Rnew
    B_new ← B_new + R_new
    R_new ← get_next_request()
  if B_new ≠ ∅ then
    prefill(B_new)   ] run all NEW prefills
    B ← B + B_new
  else
    decode(B)  → only run decodes once we've finished new prefills
  B ← filter_finished_requests(B)
```

PREFILL PRIORITIZING: at each iteration, run all prefills we can

↳ this makes TBT longer because we can have "generation stalls":

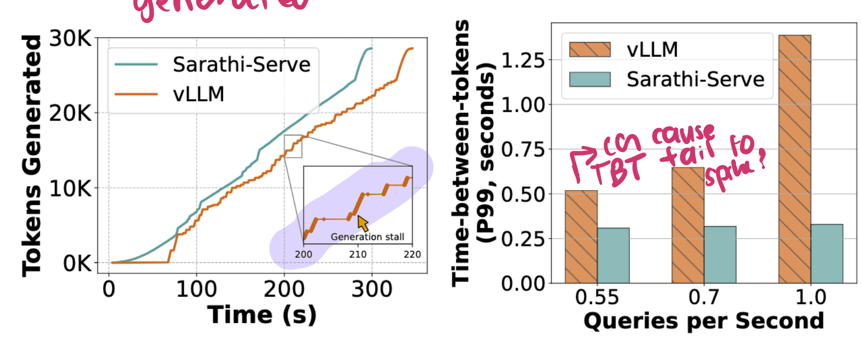
- before executing line 12 (decode), we see if we have new prefills to run and run those first.

→ so there is a fundamental tradeoff btwn TBT latency and Throughput in existing schedulers:

- at each iteration, we have the choice to:
  1. run existing requests decode cycles - prioritizes TBT at expense of TBT
  2. run prefills - which will "stall ongoing decodes"

- what does this stalling look like?
  - consider vllm in the graph below

↱ when decodes are interrupted, there will be a "stall" in new token generated



(a) Generation stall. (b) High tail latency.

→ SARATHI SERVE / CHUNKED PREFILLS.

→ paper that analyzes existing inference system and shows mismatch in tput / latency

→ proposes 2 techniques to solve:

1) chunked prefills

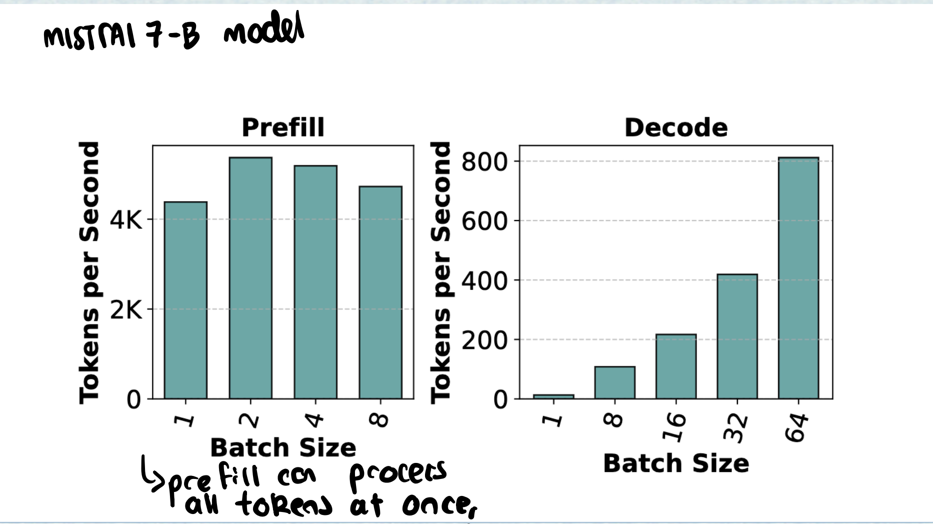
2) stall-free batching

- we will try to build up these techniques

· 3 (related) observations:

→ OBSERVATION 1:

- batching boosts throughput for decode, but not so much for prefill



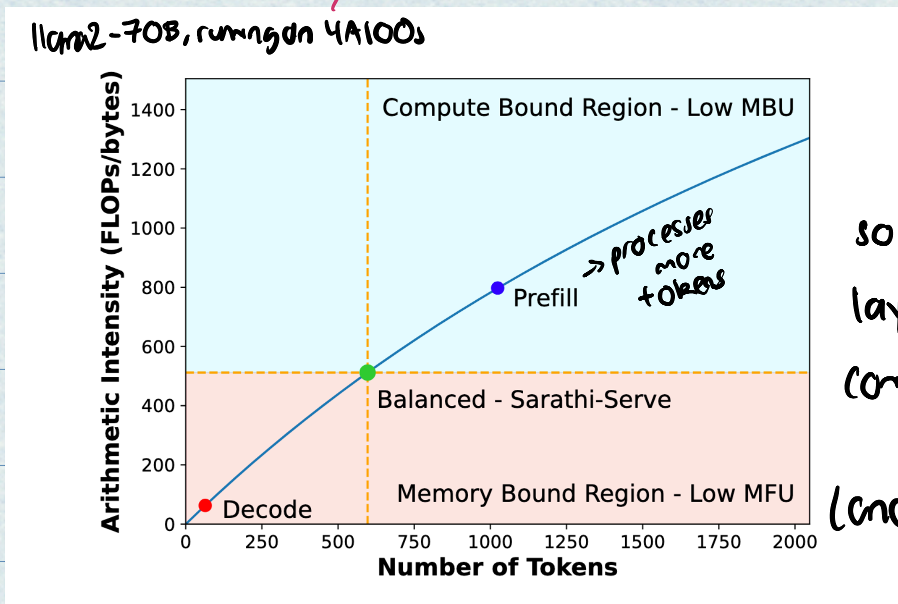
↳ prefill can process all tokens at once, even batch size 1 is highly parallelizable

· To get more tput in decode, we need to run more decodes.

Observation 2:

- Decode has low AI (memory bound, low compute utilization)
- Prefill has high AI (compute bound, low memory bw utilization)
- neither at ridge point

these schemes use pipeline parallelism - more thoughts in paper



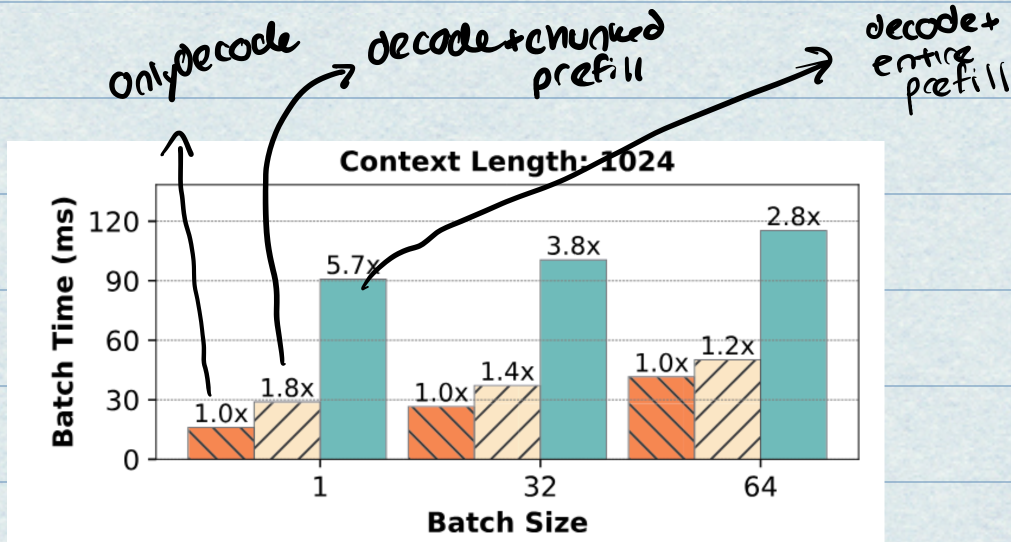
- this graph JUST focuses on linear layers - so cost of fetching layer weights vs. doing computation on linear layers (another graph in paper shows linear ops are largest time spent)

- when we are in memory bound regions (e.g., decode) - we can do more compute for same data fetched for free
  - so *more tokens can be processed along w/ decodes w/o ruining TBT for ongoing decodes

- OBSERVATION 3:
  - as we discussed before, existing systems choose at each iteration to run new prefills (to inc. throughput) OR finish existing requests
  - running prefills can stall decodes

IDEA 1: chunked prefill

- when we have extra capacity from decodes, we'd like to do more work (decode has low AI, and is memory bound)
- if we try to add in a ENTIRE prefill - and we schedule a single iteration w/ existing decodes and a full prefill, latency will go up
- KEY IDEA: lets CHUNK the prefill!
  ↳ and execute small blocks at a time



* we can execute a small portion of prefill along w/ decodes. to increase throughput w/o affecting latency that much

IDEA 2: STALL-Free Batches

↳ w/ chunked prefills, how do we schedule each iteration?

paper has more details on calculating this, but we can think of token as additional capacity until we become compute bound

```
Input: T_max, Application TBT SLO.
Initialize token_budget, τ ← compute_token_buget(T_max)
Initialize batch_num_tokens, n_t ← 0
Initialize current batch B ← ∅
while True do
    for R in B do
        if is_prefill_complete(R) then
            n_t ← n_t + 1
```

→ 1st decode

```
    for R in B do
        if not is_prefill_complete(R) then
            c ← get_next_chunk_size(R, τ, n_t)
            n_t ← n_t + c
```

→ add next prefill for existing requests

```
    R_new ← get_next_request()
    while can_allocate_request(R_new) ∧ n_t < τ do
        c ← get_next_chunk_size(R_new, τ, n_t)
        if c > 0 then
            n_t ← n_t + c
            B ← R_new
        else
            break

    process_hybrid_batch(B)
    B ← filter_finished_requests(B)
    n_t ← 0
```

→ only add new prefill chunks after
(a) existing decodes
(b) ongoing chunked prefills

→ prioritizes ongoing decodes: Always schedule these 1st to run

→ THEN schedule next chunk of existing batches

→ THEN only admit new requests

*process hybrid batch.

- can execute batch of decodes w/ chunked prefills at same time
- previous systems were only executing a batch of decodes OR batch of prefills
- paper has ablations on hybrid batch technique

* some related themes of work:

- because prefill and decode behave so differently, why not put them on dif. GPUs? scheduling will be easier
  - ↳ can send KV cache in between
- lets cache KV cache in CPU memory (when it needs to be reused) - e.g. documents and transfer it
  - ↳ but compress it in a smart way so its not so large to send