ECE7115 ~~Multimodal VLM~~ LLM

10. Inference

Spring 2026

Namhyuk Ahn, Inha University

Last Week

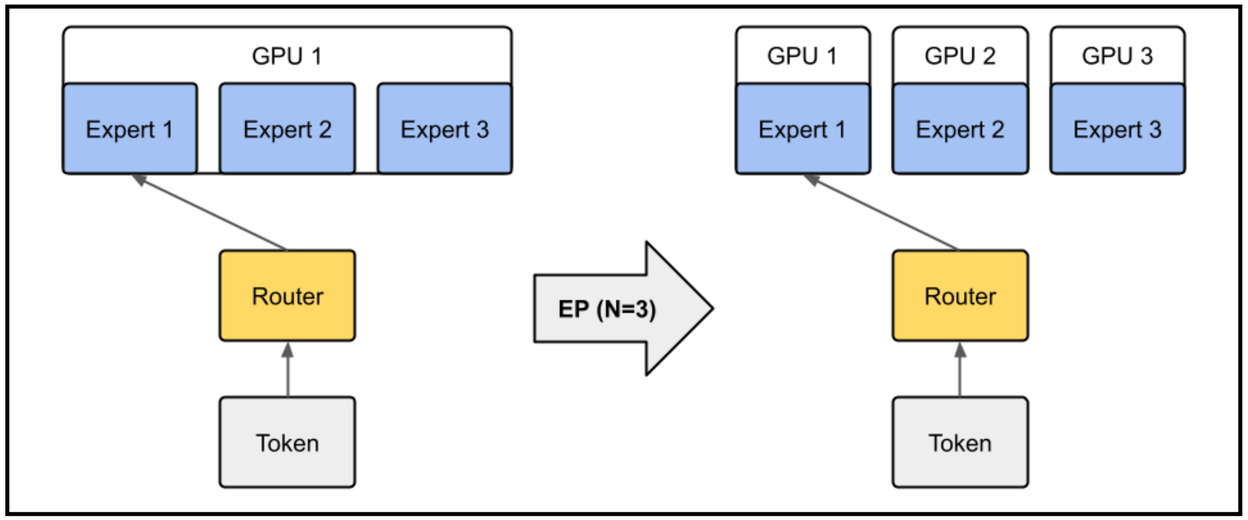
- What are each of the parallelism primitives good for?

	Sync. Overhead	Memory	Bandwidth	Batch size
DDP/ZeRO-1	Per-batch	No scaling	2x # param.	Linear
FSDP (ZeRO-3)	3x per-FSDP block	Linear	3x # param.	Linear
PP	Per-pipeline	Linear	bsh	Linear
TP + SP	2x Transformer block	Linear	8bsh per-layer (all-reduce)	No impact

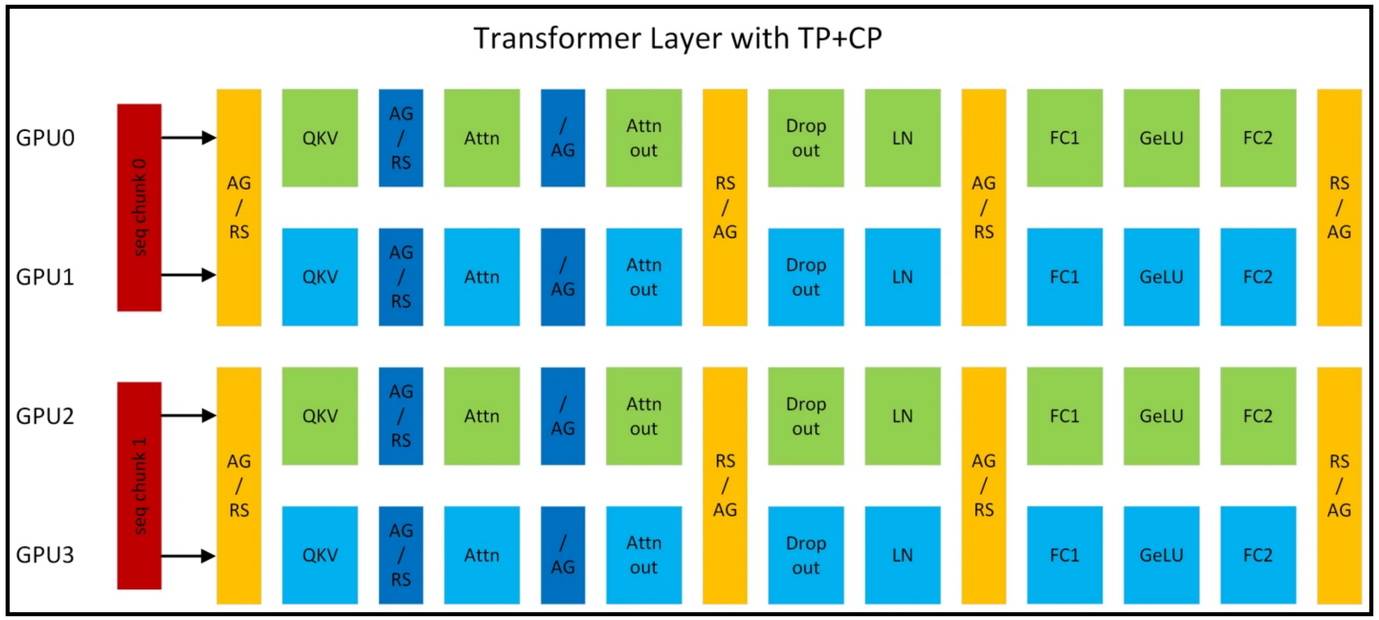
- Have to balance limited resource – memory, bandwidth, batch size

Last Week

- Expert parallelism (EP)
  - Splits experts in MoE models



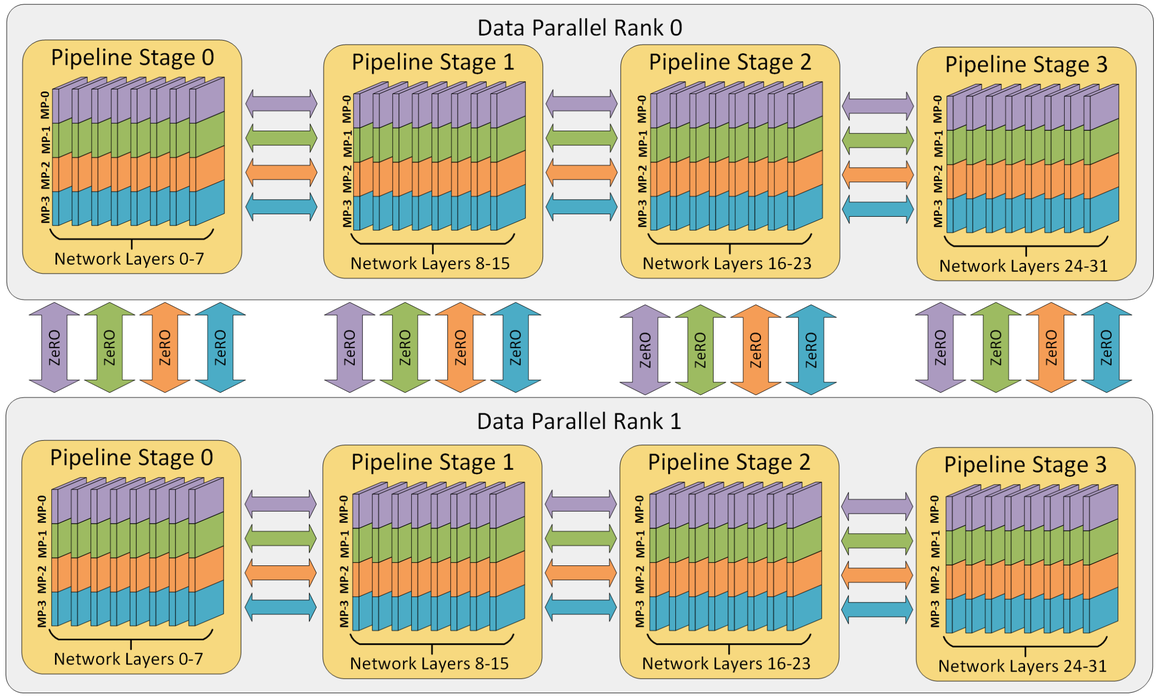
- Context parallelism (CP)
  - Split a long sequence into multiple GPUs



Last Week

- Simple rules of thumb from the literature
- 1. Until your model fits in memory
  - TP up to GPUs / machine
  - PP across machines
  - (Or use ZeRO-3, depending on BW)
- 2. Then until you run out of GPUs
  - Scale the rest of the way with DP

- If your batch size is small…
  gradient accumulate to trade higher
  batch sizes for better communication efficiency

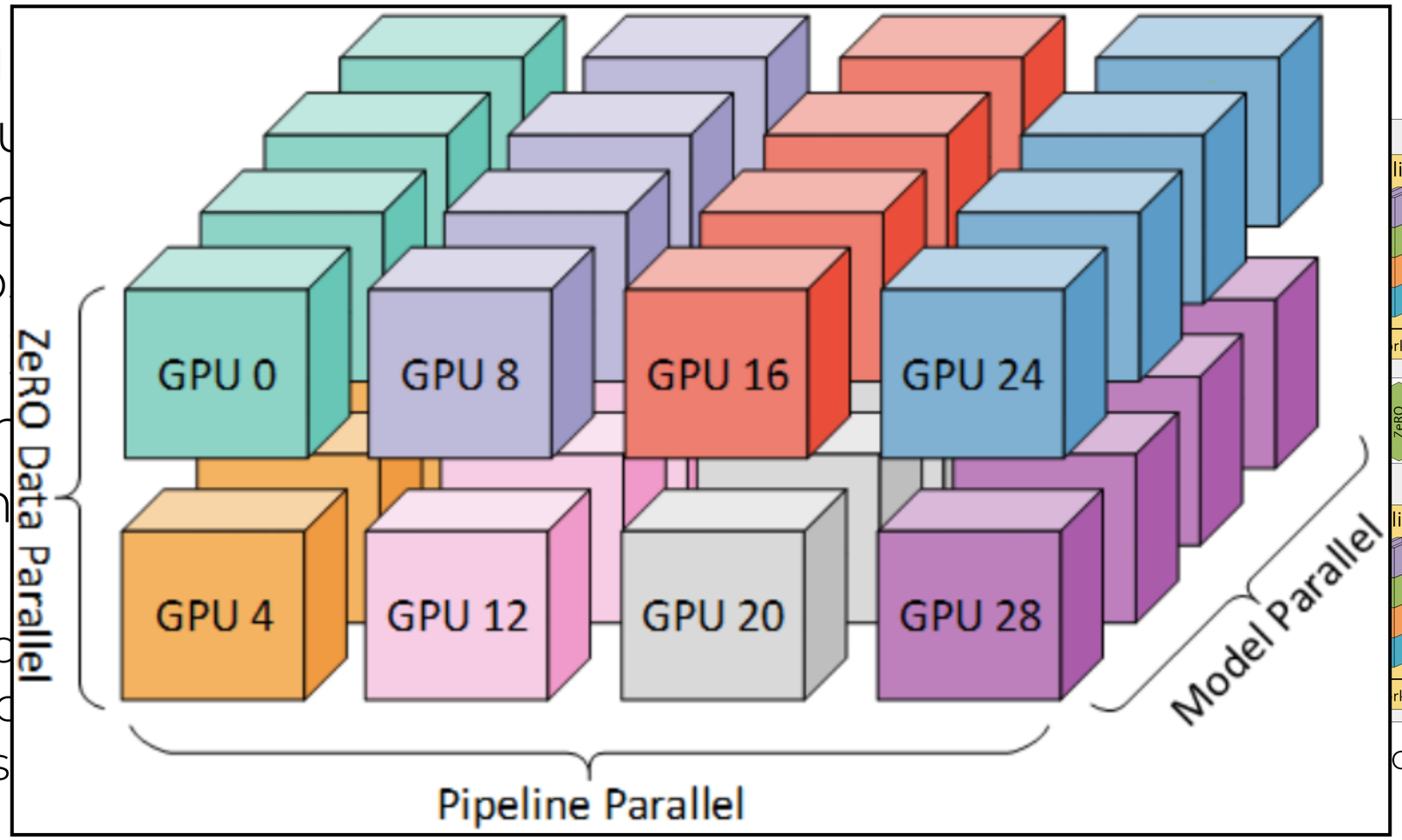


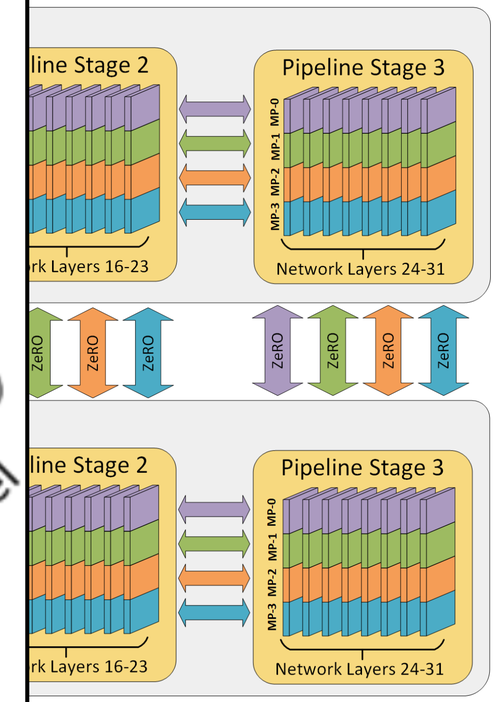
https://www.deepspeed.ai/tutorials/pipeline/

Last Week

- Simple ru
- 1. Until you
  - TP up to
  - PP acro
  - (Or use
- 2. Then ur
  - Scale th

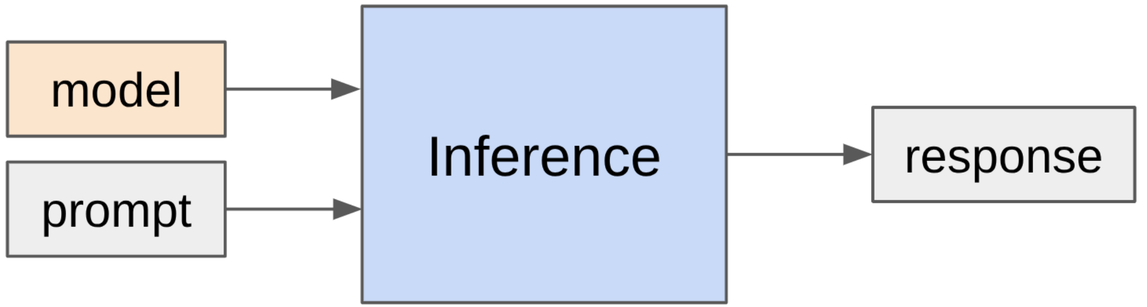
- If your batc
  gradient ac
  batch sizes





d.ai/tutorials/pipeline/

Inference



- Inference shows up in many places
  - Actual use (chatbots, code generation, agents, API, etc)
  - Model evaluation (e.g., on instruction following)
  - RL-based training (sample many generations, then apply score)

- Why inference efficiency matters?
  - Training is one-time cost, inference is repeated many times
  - OpenAI processes ~8.6T tokens per day
  - For reference, DeepSeek-V4 was trained on 32T tokens
- At scale, inference can consume training-scale amounts of compute

LLM Inference is Token Generation

- In LLMs, every generated token costs compute
  - Different use cases have very different token economics
  - Chatbots
    - Most generated tokens are meant for human consumption
    - Latency matters, but humans are relatively slow.

  - Agents
    - Agents generate many tokens that humans may never see
    - Internal tokens can dominate visible tokens
    - Inference cost can scale with task difficulty, not just answer length

Why LLM Serving Is Hard

- LLM serving is not just "running a model faster"
- LLM serving ≠ Ordinary Inference

Ordinary ML Inference	LLM Serving
One input	Prompt + generated tokens
One forward pass	Many forward passes
Fixed output size	Variable output length
Stateless execution	Stateful KV cache
Simple batching	Dynamic scheduling

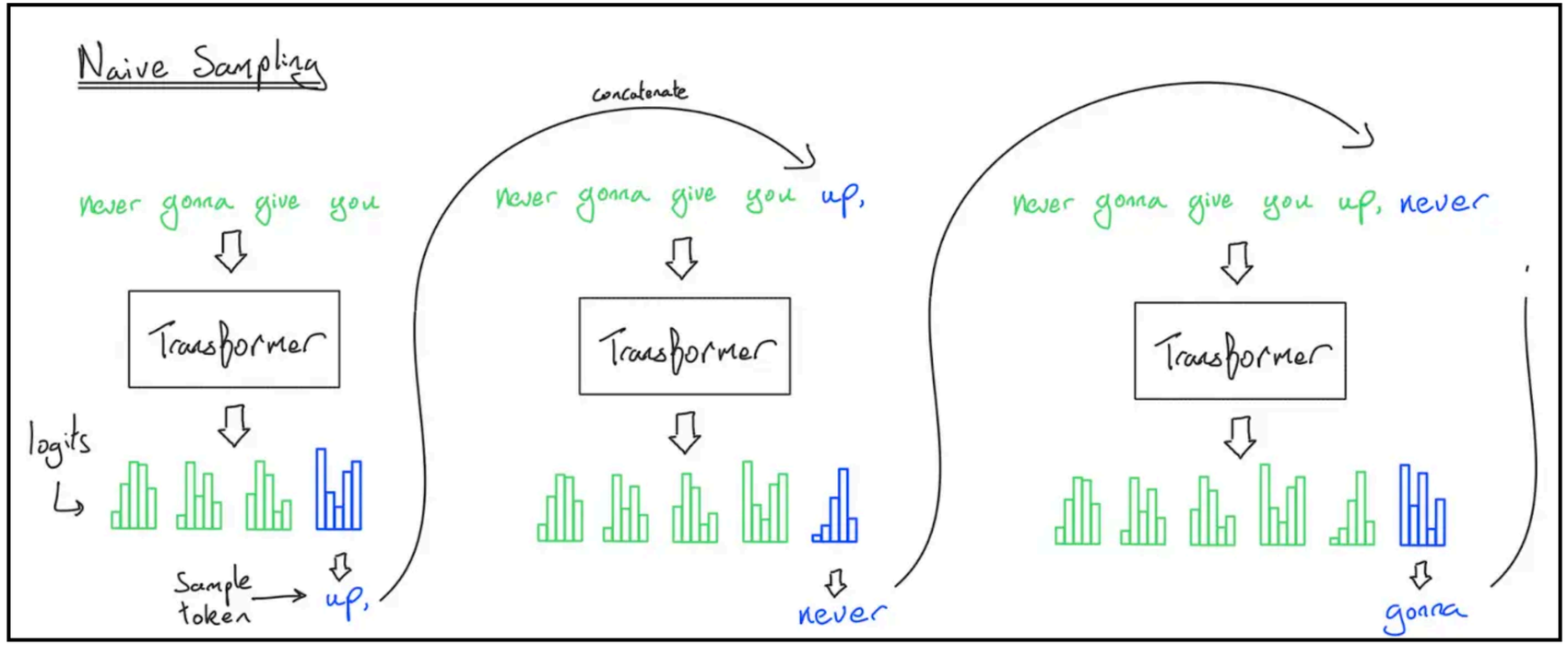
Why LLM Serving Is Hard

- One request has two very different phases
  - Prefill stage and decode stage

- LLM workloads are highly variable
  - Input and output lengths vary depending on the requested job

Image credit: https://jax-ml.github.io/scaling-book/inference/

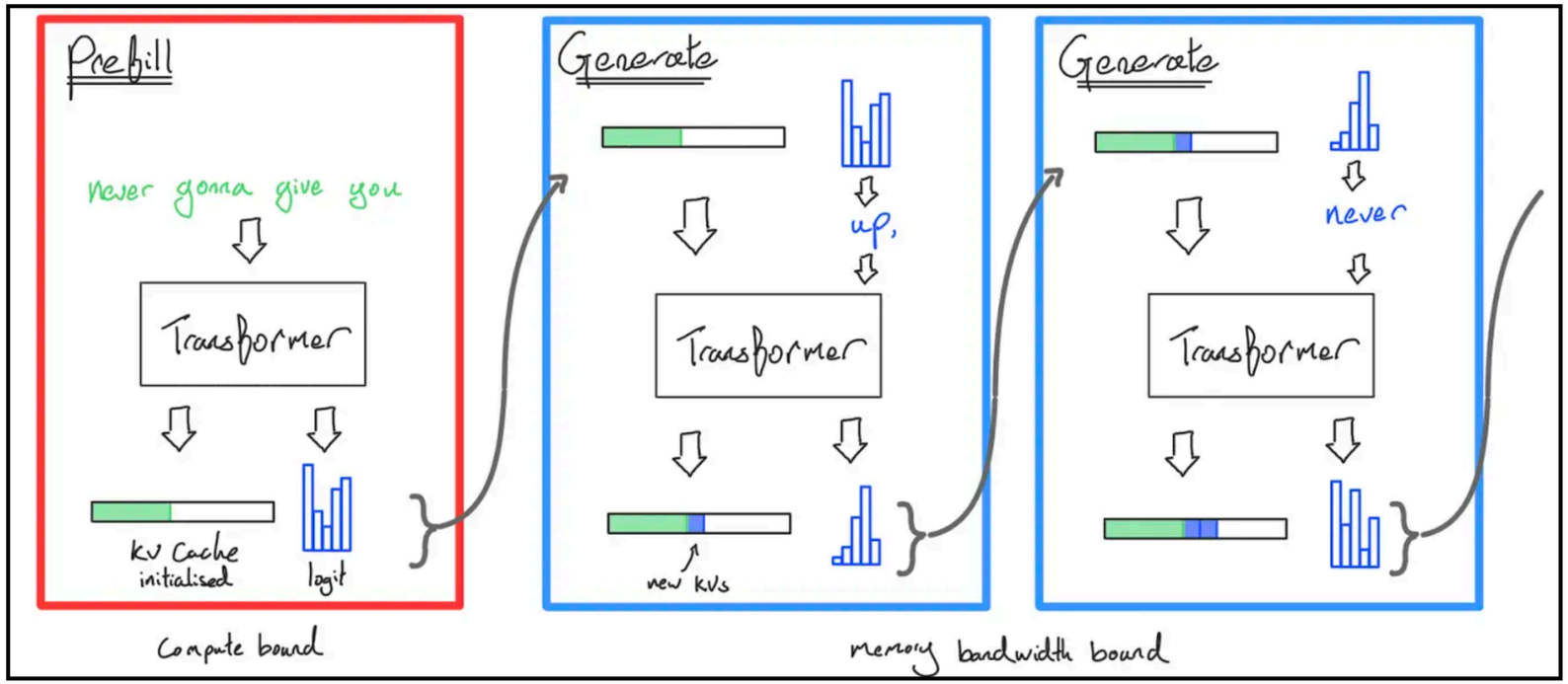
Recap: Naive Inference



- Naive inference: to generate each token, feed history into Transformer
- Complexity: generating T tokens requires $O(T^3)$ FLOPs
  (one feedforward pass is $O(T^2)$)

Image credit: https://jax-ml.github.io/scaling-book/inference/

Recap: Prefill → Decode



- Prefill: given a prompt, encode into vectors (parallelizable like in training)
- Decode (generation): generate new response tokens (sequential)

Prefill vs. Decode

- Prefill: process N input prompt tokens at once
  - Large GEMM (GEneral Matrix Multiplication; matrix × matrix)
  - Fills the KV cache in a single forward pass
  - Compute-bound

- Decode: generate tokens one by one auto-regressively
  - Repeated small GEMV (matrix × vector)
  - Reads all weights at every step
  - Memory-bound

- It's the same model, but the hardware workload is completely different

Open-Sourced Inference Packages

- vLLM (Berkeley)
  - Pioneered PagedAttention, popular and good default
- SGLang (Berkeley)
  - Pioneered RadixAttention, good for agentic workloads
- TensorRT-LLM (NVIDIA)
  - Highly optimized for GPUs
- llama.cpp
  - C++ only, supports CPU inference, runs locally

Lecture Overview

- Sampling

- Serving Metrics
- (Recap) Reducing KV-Cache

- Serving Systems (mostly from vLLM)
  - Batching, Paged Attention
  - Speculative Decoding
  - Quantization

Sampling

Token Sampling

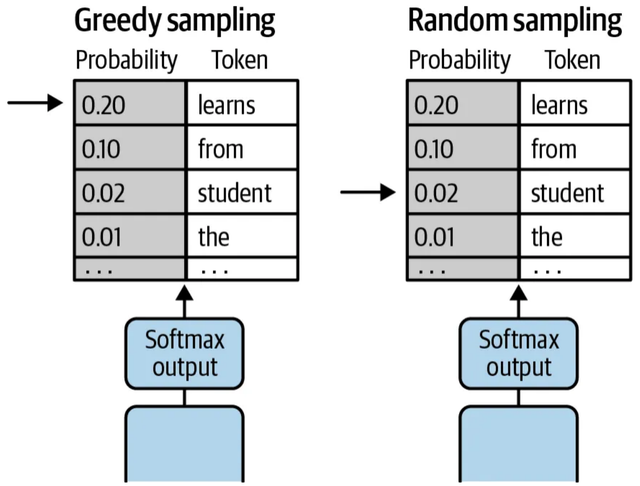
- One step decoding takes a sequence $x_{1,\ldots,t}$ and return a token x_{t+1}

$$P(x_{t+1} = i \mid x_{1\ldots t}) = \frac{\exp(v_i)}{\sum_j \exp(v_j)}$$

$$v = \mathrm{TransformerLM}(x_{1\ldots t})_t \in \mathbb{R}^{\texttt{vocab\_size}}$$

- Greedy sampling
  - Do argmax to the Softmax logits, fully deterministic, very simple
  - But repetition loops, no diversity

- Random sampling
  - Random selection w.r.t Softmax logits

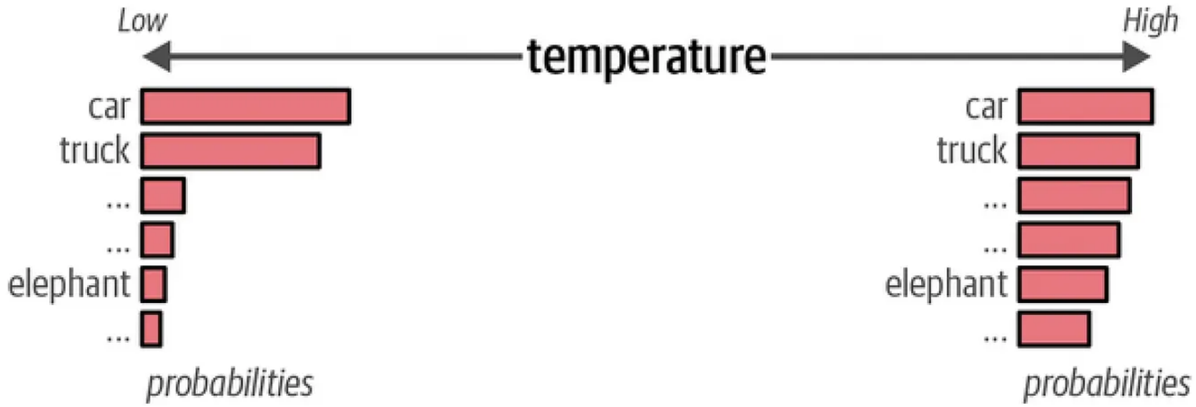


Temperature Scaling

- Temperature scaling modifies Softmax with a temperature parameter

$$\text{softmax}(v, \tau)_i = \frac{\exp(v_i/\tau)}{\sum_{j=1}^{|\text{vocab\_size}|} \exp(v_j/\tau)}$$

- τ makes the trade-off between diversity and quality
  - Low τ scaling: sharper distribution, more deterministic (goes greedy)
  - High τ scaling: flatter distribution, more diversity (but more noisy)



Top-K Sampling

- Even with temperature scaling, random sampling cannot suppress low-quality (low-prob) tail tokens

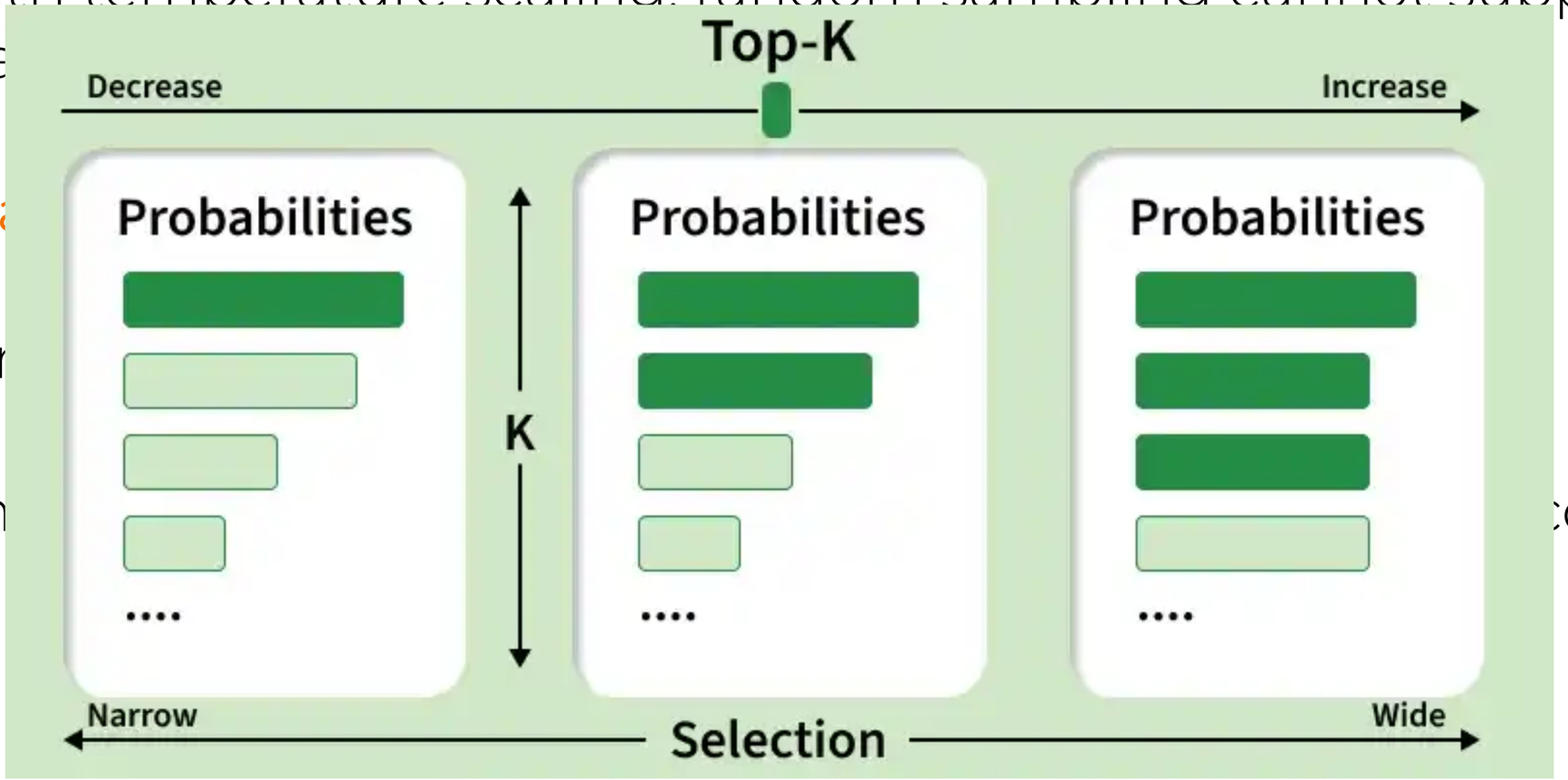
- Top-K sampling: keep top-K token logits, zero-out the rest
- Among top-k tokens, we randomly choose final token according to their (normalized) logit probability

- K=1 is the same as argmax; and is called greedy sampling (decoding)

Image credit: https://www.geeksforgeeks.org/artificial-intelligence/graph-based-semi-supervised-learning/

Top-K Sampling

- Even with temperature scaling, random sampling cannot suppress low-qua
- Top-K sa
- Among
  accordi
- K=1 is th coding)



Top-p Sampling (Nucleus Sampling)

- Top-K cutoffs K tokens, regardless of the distribution shape

- Top-p (nucleus): keep top tokens until cumulative prob. reaches p
  - Few tokens when distribution is sharp, many when flat distribution

$$P(x_{t+1} = i|q) = \begin{cases} \frac{q_i}{\sum_{j \in V(p)} q_j} & \text{if } i \in V(p) \\ 0 & \text{otherwise} \end{cases}$$

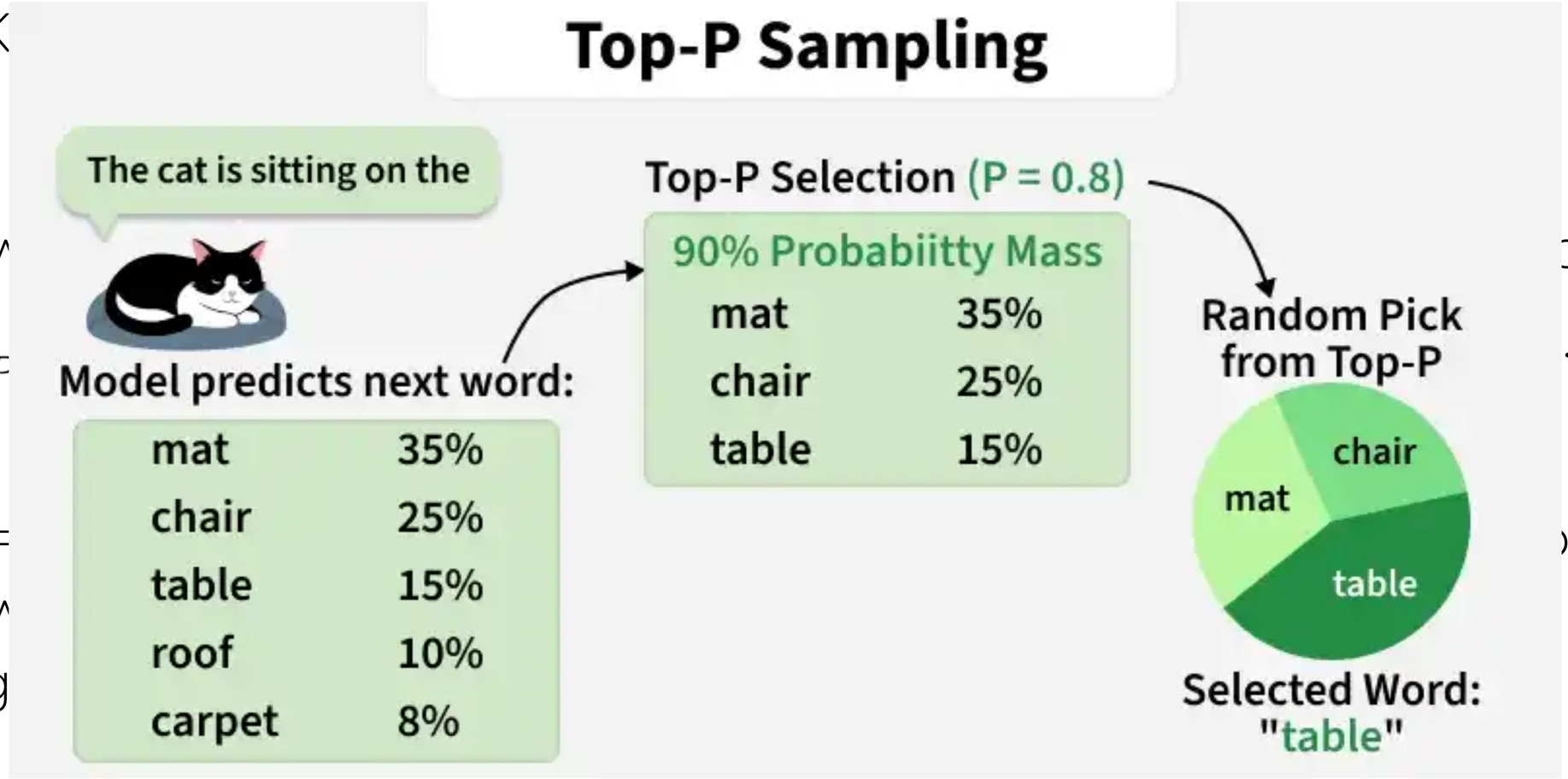
$V(p)$ is the smallest set s.t. $\sum_{j \in V(p)} q_j \geq p$.

- e.g. p=0.9: selects from words that account for 90% of the total prob.
  - Lower p: fewer choices, safer output
  - Higher p: more choices, more creative output

Image credit: https://www.geeksforgeeks.org/artificial-intelligence/graph-based-semi-supervised-learning/

Top-p Sampling (Nucleus Sampling)

- Top-K
- Top-p
  - Few
- e.g. p=
  - Low
  - Hig



Serving Metrics

User-Side Inference Metrics

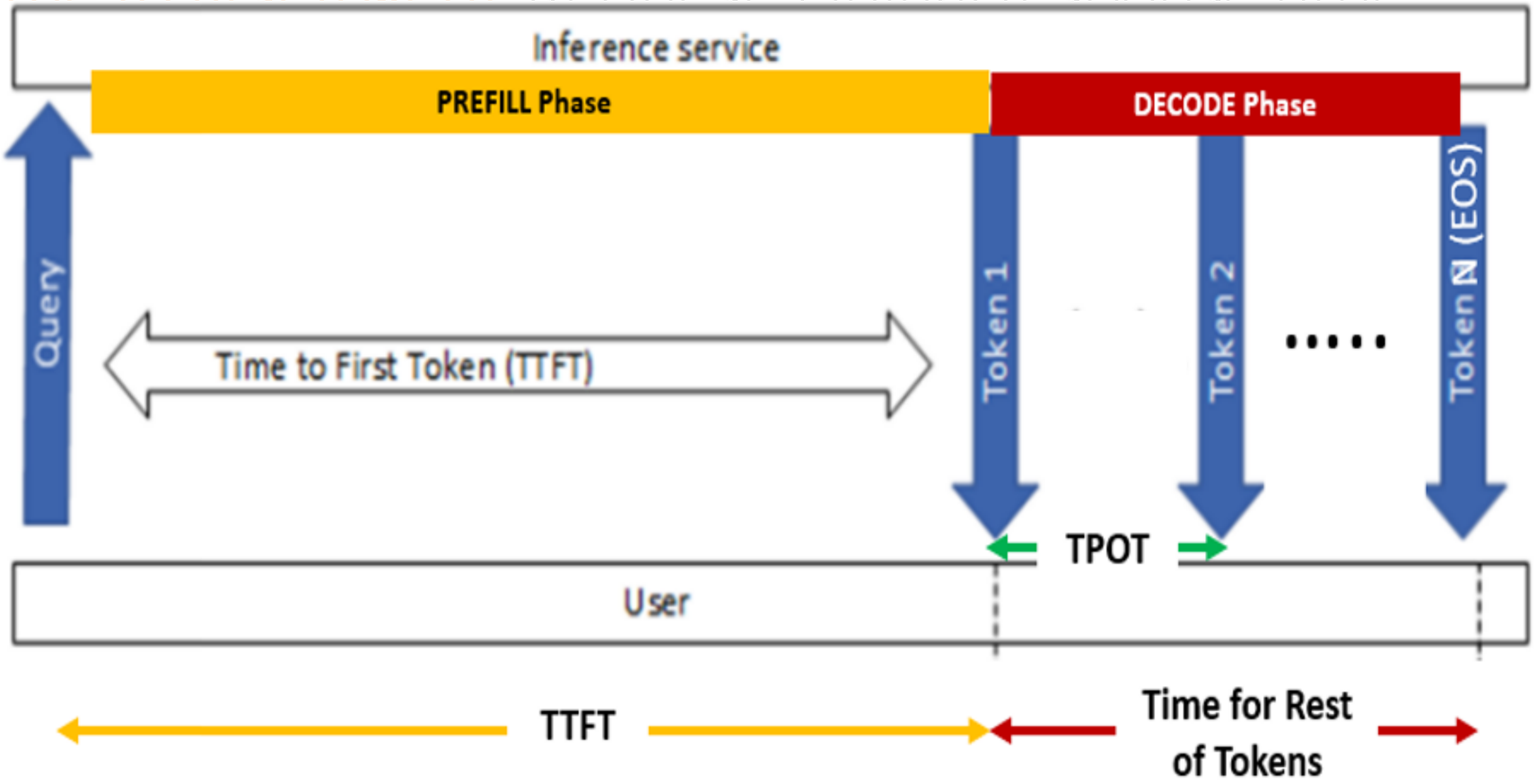
- TTFT (Time to first token): Request → initial response time
  - Determines perceived responsiveness
  - Most sensitive metric in service
  - Roughly proportional to prefill cost

- TPOT (Time per output token) or ITL (Inter-token latency)
  - Interval between tokens, smoothness of streaming
  - Proportional to decode cost

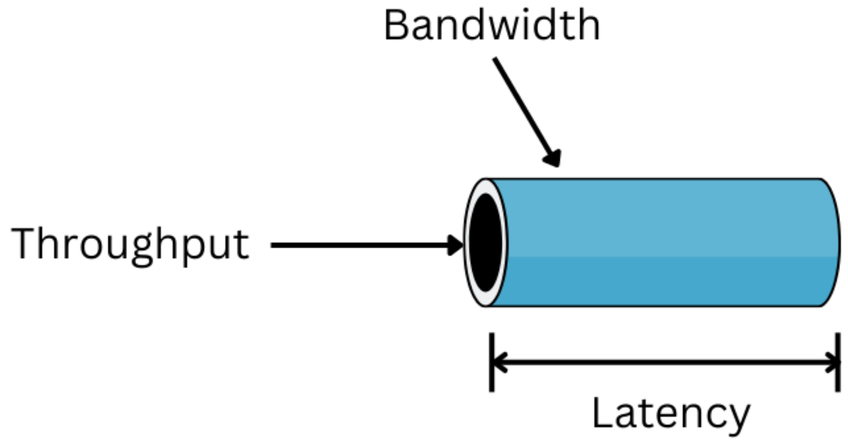
- End-to-end latency = TTFT + TPOT × output_len

User-Side Inference Metrics

- TTFT (Time to first token): Request → initial response time
  - Dete
  - Most
  - Roug
- TPOT (
  - Inter
  - Prop
- End-to



Server-Side Inference Metrics



- Throughput: tokens/sec, requests/sec

- Latency vs. throughput trade-off
  - batch ↑ → throughput ↑, but per-request latency also ↑
  - Satisfying both metrics is the core challenge of serving

  - Naively maximizing throughput can violate SLOs
    - SLO? Service level indicator
    - (e.g. latency, throughput, error rate, availability, …)

  - → introduce Goodput

Goodput

- Good throughput: throughput of requests that satisfy the SLO
  - e.g. only count requests where TTFT < 500 ms AND TPOT < 50 ms

- Common tricks that inflate naive throughput:
  - Increasing batch raises throughput but explodes TTFT → users leave

- Goodput acts as the decision function for system design
  - It motivates techniques like Continuous batching and P/D disaggregation (which reappear later)

Cost Metrics

- $/1M tokens: ultimately GPU time × GPU price
  - $/token ≈ (GPU $/sec) / (tokens/sec per GPU)
  - Numerator: infrastructure cost (fixed)
  - Denominator: throughput (the optimization target)

- Capacity equation: capacity = throughput × utilization

- Price is ultimately governed by decode tokens/sec per GPU
- → Optimizing decode (inference) cost matters most

Reducing KV-Cache

Recap: Arithmetic Intensity

- Operational efficiency of an algorithm relative to data movement
  - $\mathbf{I}$ = Total Operations (FLOPs) / Total Communication (Bytes)
  - Higher $\mathbf{I}$: Efficient; high volume of computation per data fetch
  - Lower $\mathbf{I}$: Inefficient; performance is bottlenecked by data movement

- Peak arithmetic intensity (ridge point)
  - Every hardware architecture has its own optimal threshold $\mathbf{I}_{\mathbf{ridge}}$
    - e.g. H100 has 295 FLOPs/Byte
  - $\mathbf{I} < \mathbf{I}_{\mathbf{ridge}}$: Comm.-bound (perf. limited by memory bandwidth)
  - $\mathbf{I} > \mathbf{I}_{\mathbf{ridge}}$: Compute-bound (fully utilizing compute capacity)

Recap: Matmul Example

- Setup: matmul **`X`** (**`[BxD]`**) and **`W`** (**`[DxF]`**) and produce **`Y`**
- Communication (Bytes)
  - Load **`X,W`**: 2 x B x D+ 2 x D x F / Save **`Y`**: 2 x B x F
  - Total: (2 x B x D)+ (2 x D x F) + (2 x B x F)

- Compute (FLOPs)
  - Total: 2 x B x D x F (2 for multiplication and addition)

- Arithmetic intensity
  - I = (2 x B x D x F) / ((2 x B x D)+ (2 x D x F) + (2 x B x F))
  - I = B (if we assume B is much less than D and F)

Recap: Matmul Example

- Arithmetic intensity
  - $I = B$
- Peak arithmetic intensity of H100
  - I_{ridge} = 295 (FLOP/s = 989e12, memory bandwidth = 3.36e12)
- Conclusion: compute-limited iff B > 295
  - Why? in matmul, elems in row and col are reused multiple times
- But extreme case; B = 1, then I = 1 (memory-limited)
  - Why? read D x F matrix, perform only 2xDxF FLOPs

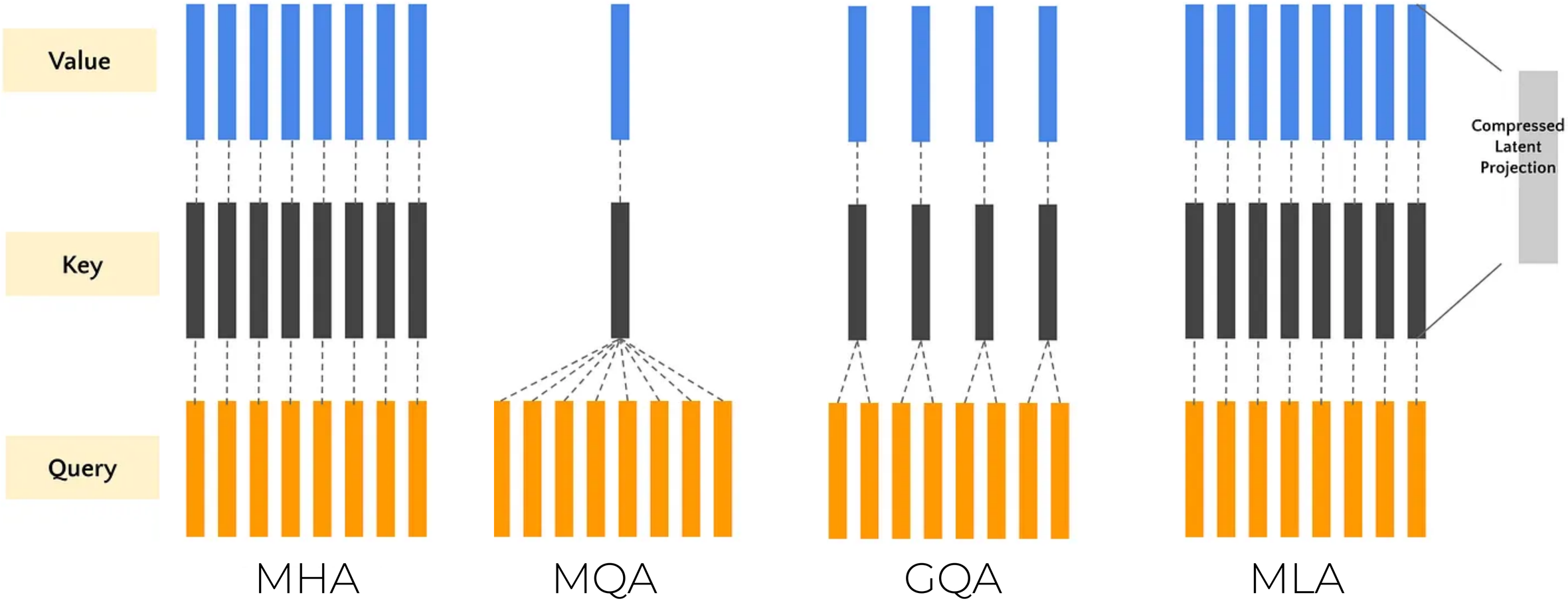
Recap: KV-Cache

2 x precision x KV dim x # heads x # layers x # seq. length

- For example, DeepSeekR1 / V3
  - 2 = key/value
  - precision = 2 bytes per elems
  - KV dim = d_k, d_v = 128
  - # heads = 128
  - # layers = 61
  - # seq. length = 32,768 tokens

  - Total required memory for just KV-cache is 131 GB!

Recap: Reducing KV-Cache



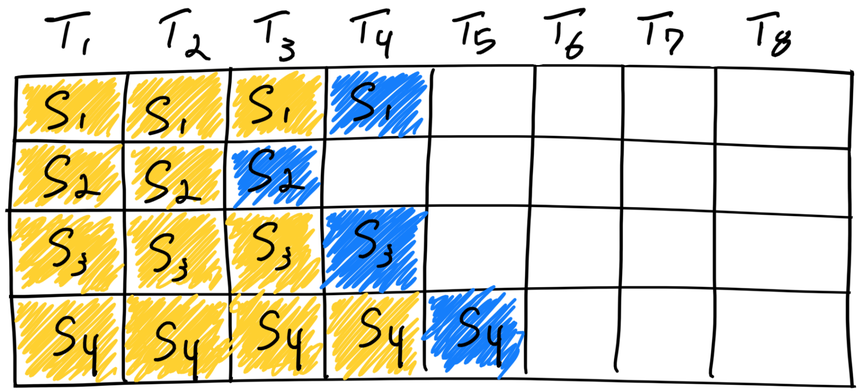
Serving Systems - Batching

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

Static Batching

- Prefill stage is compute-bound but decode is memory-bound
  - Larger batch size makes the system efficient

- Static batching
  - Bundle arrived requests and process them together



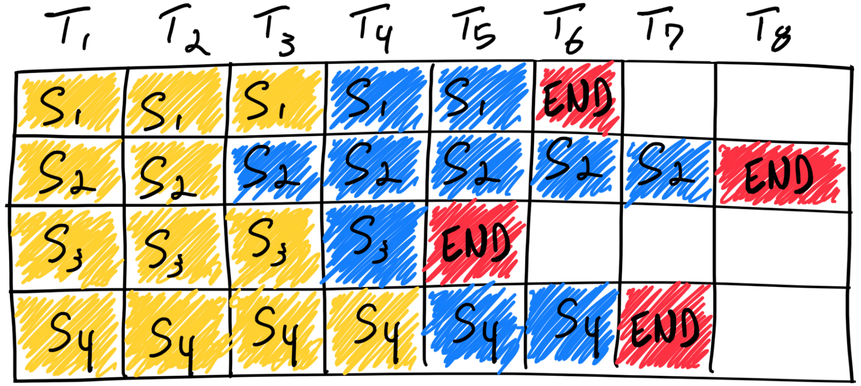


Image credit: https://www.anyscale.com/blog/continuous-batching-llm-inference

Static Batching

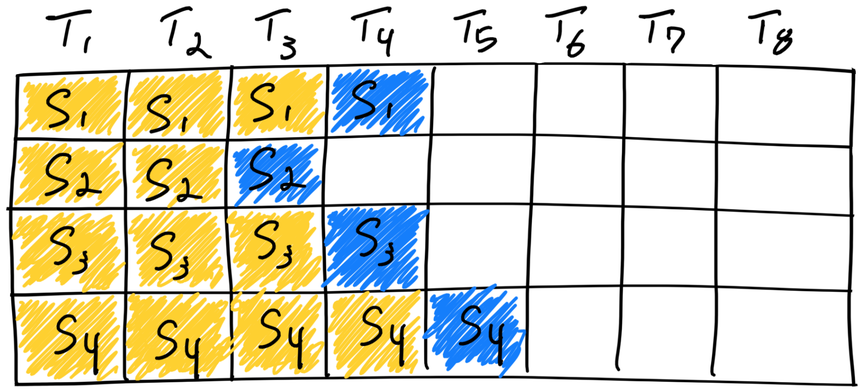
- Limitations of static batching
  - Output lengths vary per request
    - → Everyone waits for the longest one to finish
  - New requests arriving mid-batch wait for the next batch
    - → TTFT explodes
  - Slots from finished short requests sit idle, wasting GPU

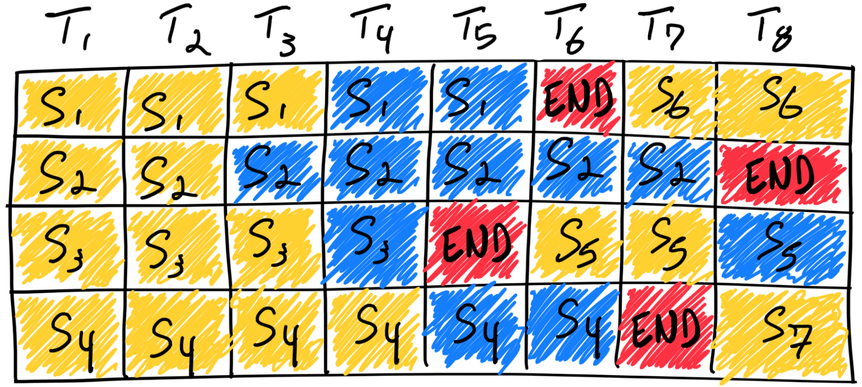
Image credit: https://www.anyscale.com/blog/continuous-batching-llm-inference

Continuous Batching [Yu+ 2022]

- Refresh batch membership at every decode iteration
  - Scheduling occurs at the decoding-iter level, not at the request level
  - Finished requests leave immediately
  - New requests join immediately (after prefill)

  - No idle slots, GPU runs full → serving throughput ↑
  - Short requests get an immediate response → serving goodput ↑





Continuous Batching [Yu+ 2022]

- But implementing continuous batching is tricky due to KV-cache

- Why?
  - Each request has a different context length
  - KV-cache grows by one token at every decode step
  - Finished requests release memory
  - New requests require memory allocation

- → This creates a dynamic memory management problem

KV-Cache Management Problem

- Each request carries its own KV-cache
- KV-cache size depends on sequence length

- In continuous batching,
  - requests enter and leave dynamically
  - active sequences have different lengths
  - memory must be allocated and freed frequently

A:	200 tokens	→	201-th token	→
B:	4K tokens	→	4K+1-th token	→
C:	Request finishes	→	free KV-cache	
D:	Request enters	→	allocate new KV-cache	

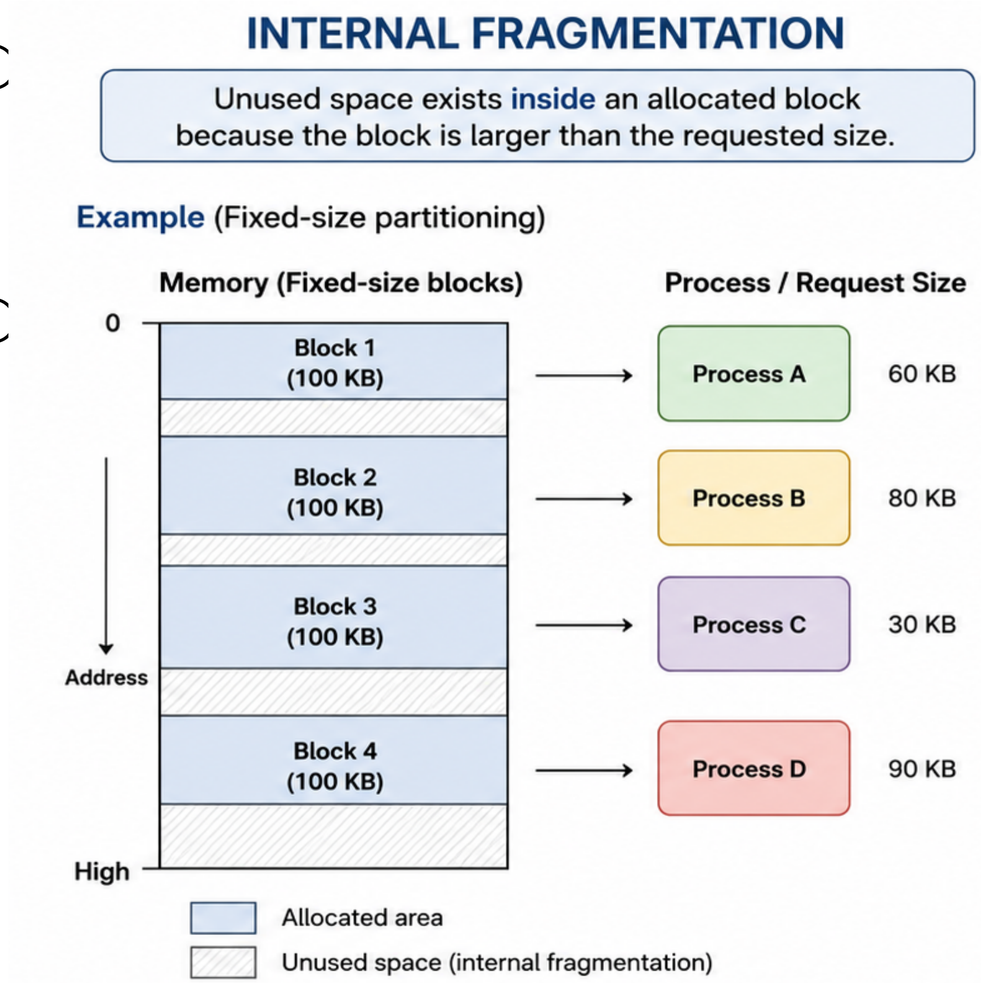
KV-Cache Allocation

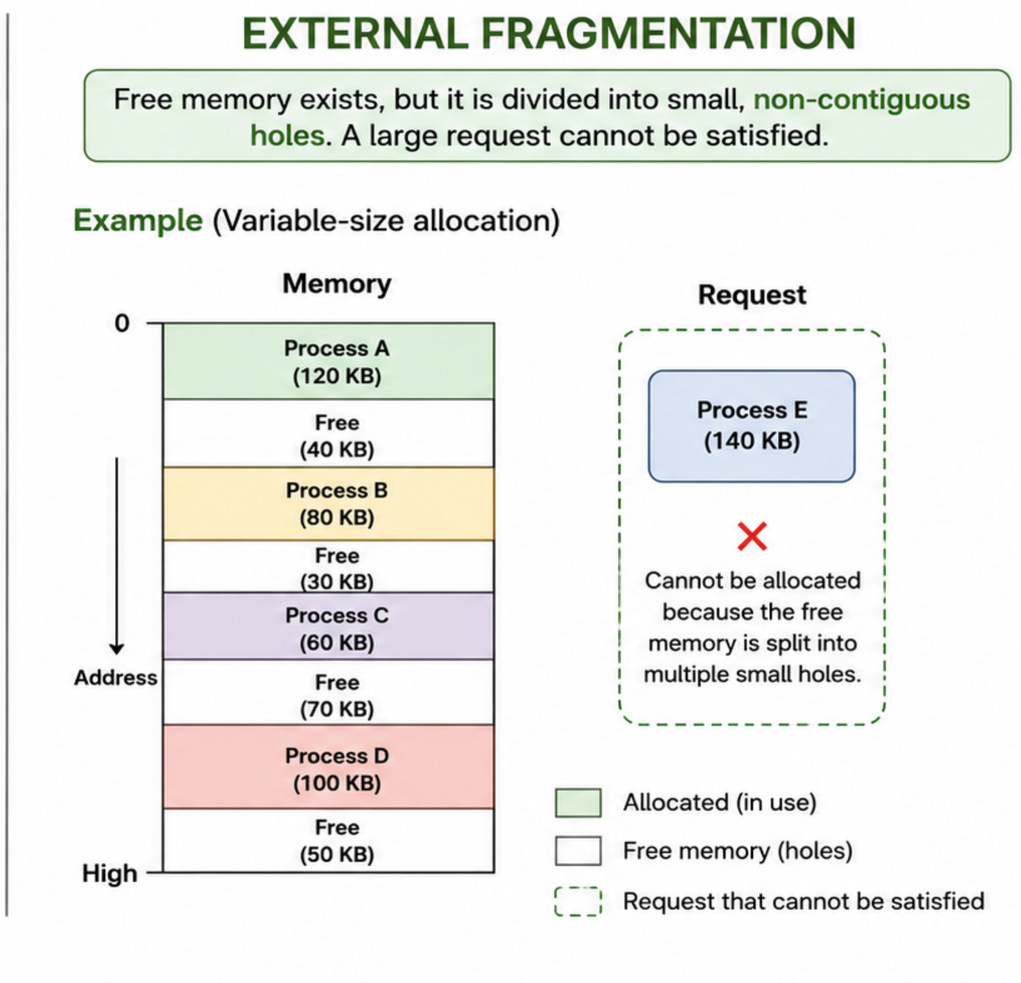
- Option 1: allocate max context length for every request
  - Simple, but huge memory waste, internal fragmentation

- Option 2: allocate contiguous memory as needed
  - Less waste, but causes external fragmentation

KV-Cache Allocation

- C
  -
- C
  -

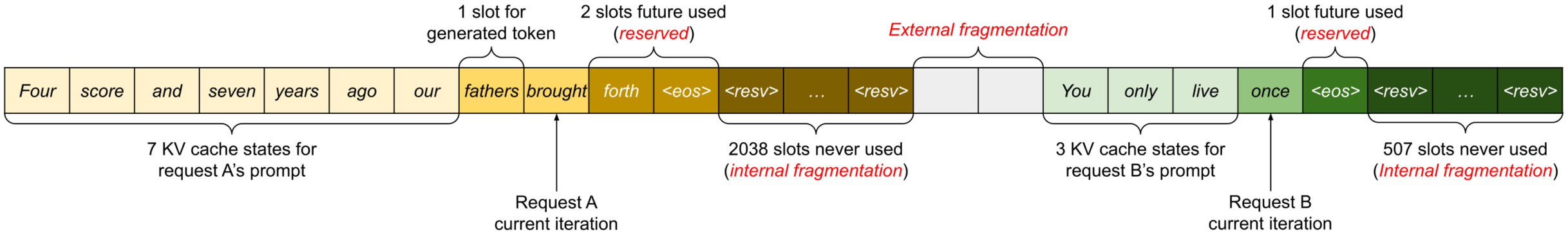




KV-Cache Allocation [Kwon+ 2023]

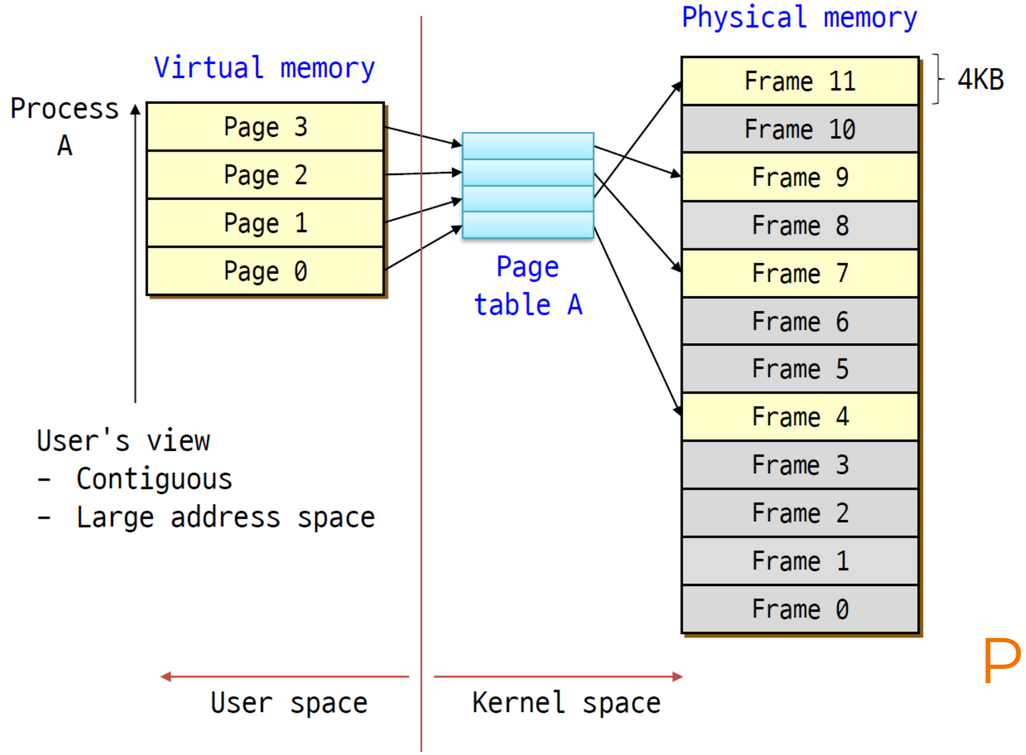
- Option 1: allocate max context length for every request
  - Simple, but huge memory waste, internal fragmentation

- Option 2: allocate contiguous memory as needed
  - Less waste, but causes external fragmentation



PagedAttention [Kwon+ 2023]

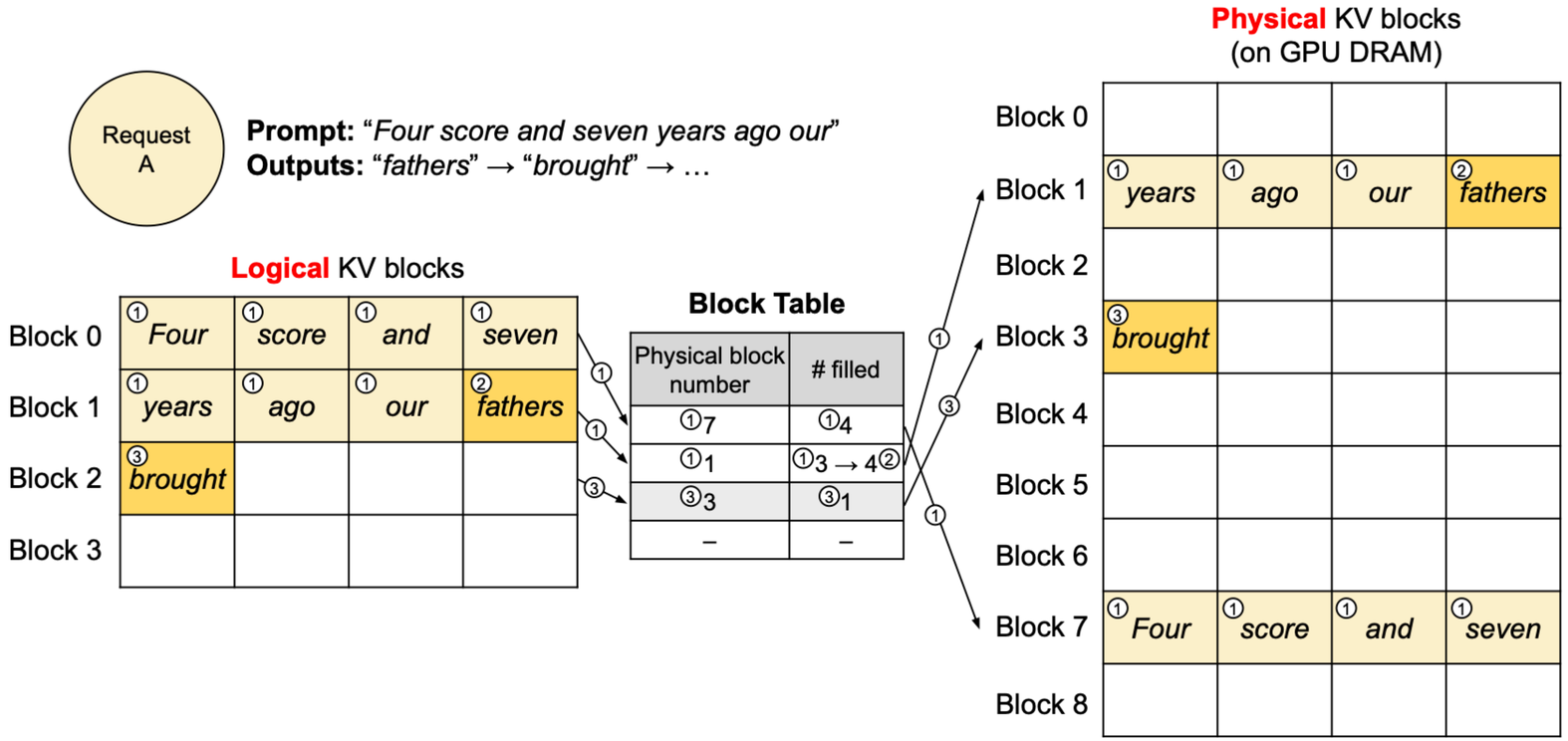
- Key idea: manage KV-cache like virtual memory (remember OS class!)
  - Split KV-cache into fixed-size blocks
  - Logical cache does not need to be contiguous in physical memory



Paging in OS

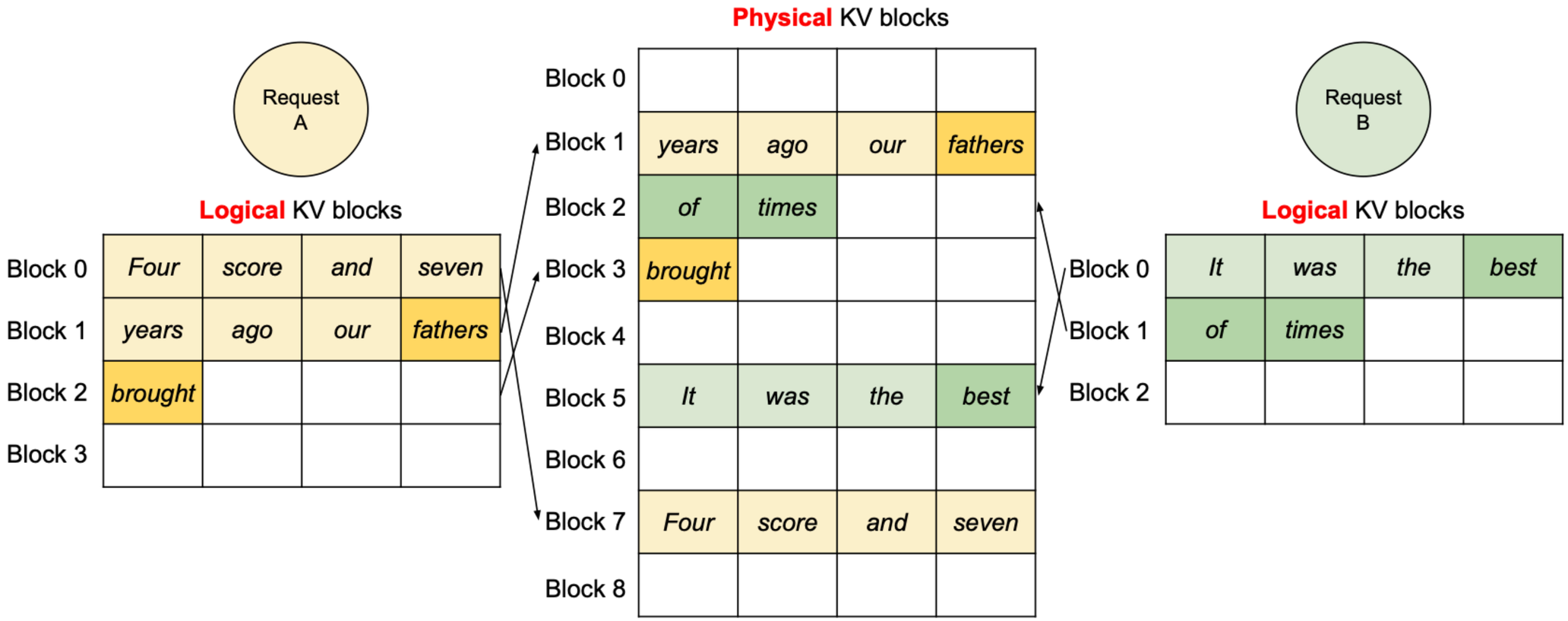
PagedAttention [Kwon+ 2023]

- Key idea: manage KV-cache like virtual memory (remember OS class!)
  - Split KV-cache into fixed-size blocks
  - Logical cache does not need to be contiguous in physical memory



PagedAttention [Kwon+ 2023]

- Storing the KV cache of two requests



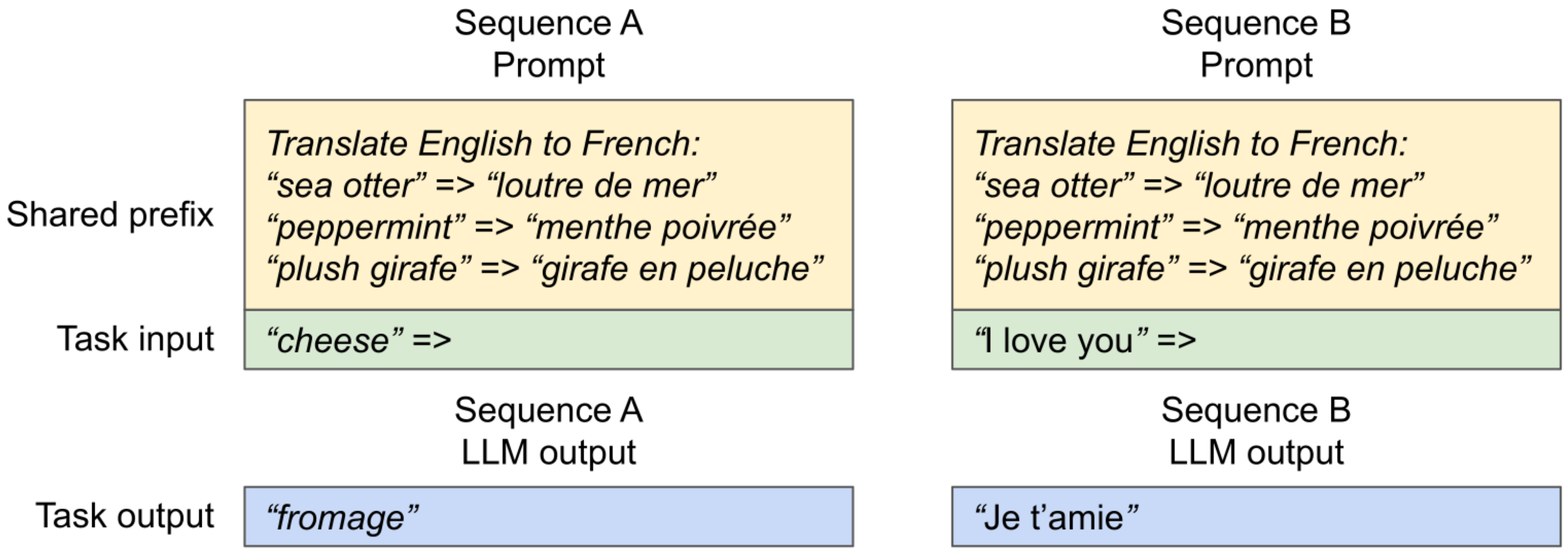
PagedAttention [Kwon+ 2023]

- Why PagedAttention works?
  - Reduces memory fragmentation (both internal and external)
  - Allows dynamic KV-cache growth
  - Reuses freed blocks from completed requests
  - Supports variable-length sequences

  - Makes continuous batching practical!

Prefix Caching

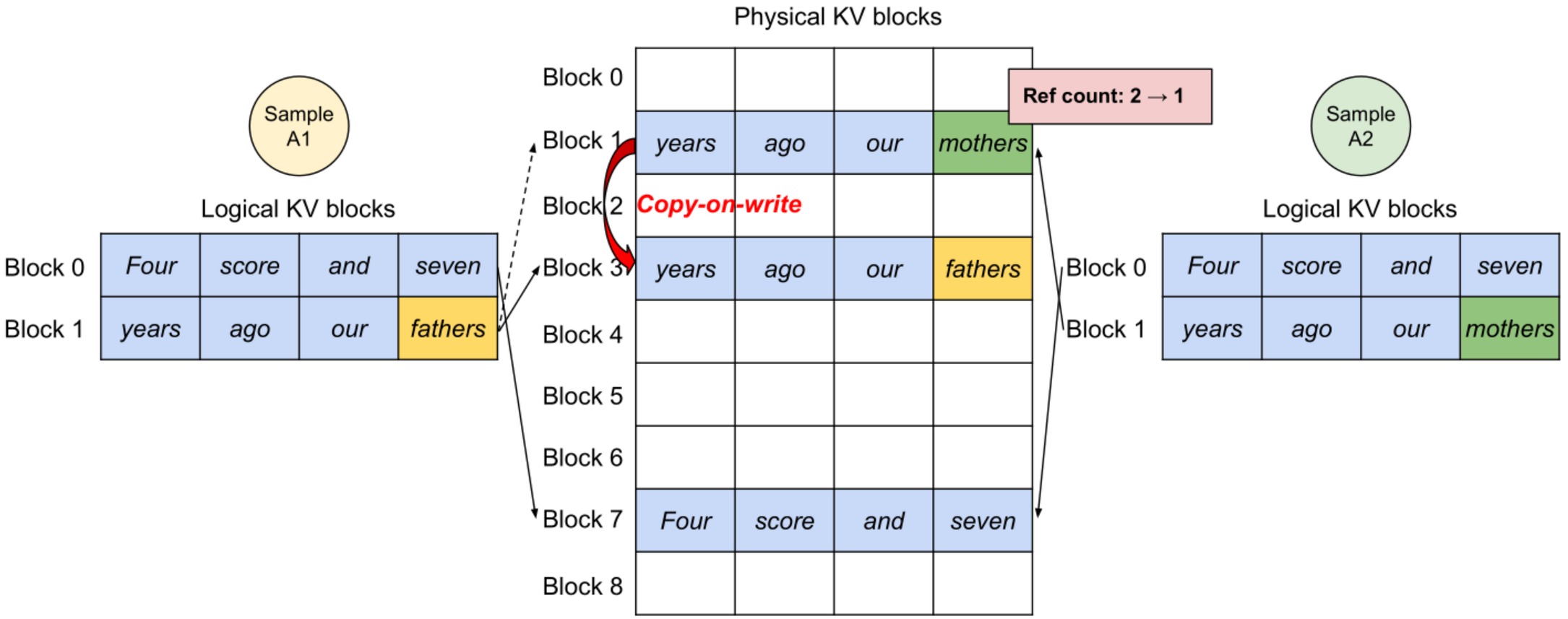
- In general, prefixes repeat very often in real traffic
  - System prompts, few-shot examples
  - The opening of a multi-turn conversation



- → Identify identical prefix KV blocks by hash → reuse them

Prefix Caching with PagedAttention [Kwon+ 2023]

- Share prefixes, copy-on-write at the block level



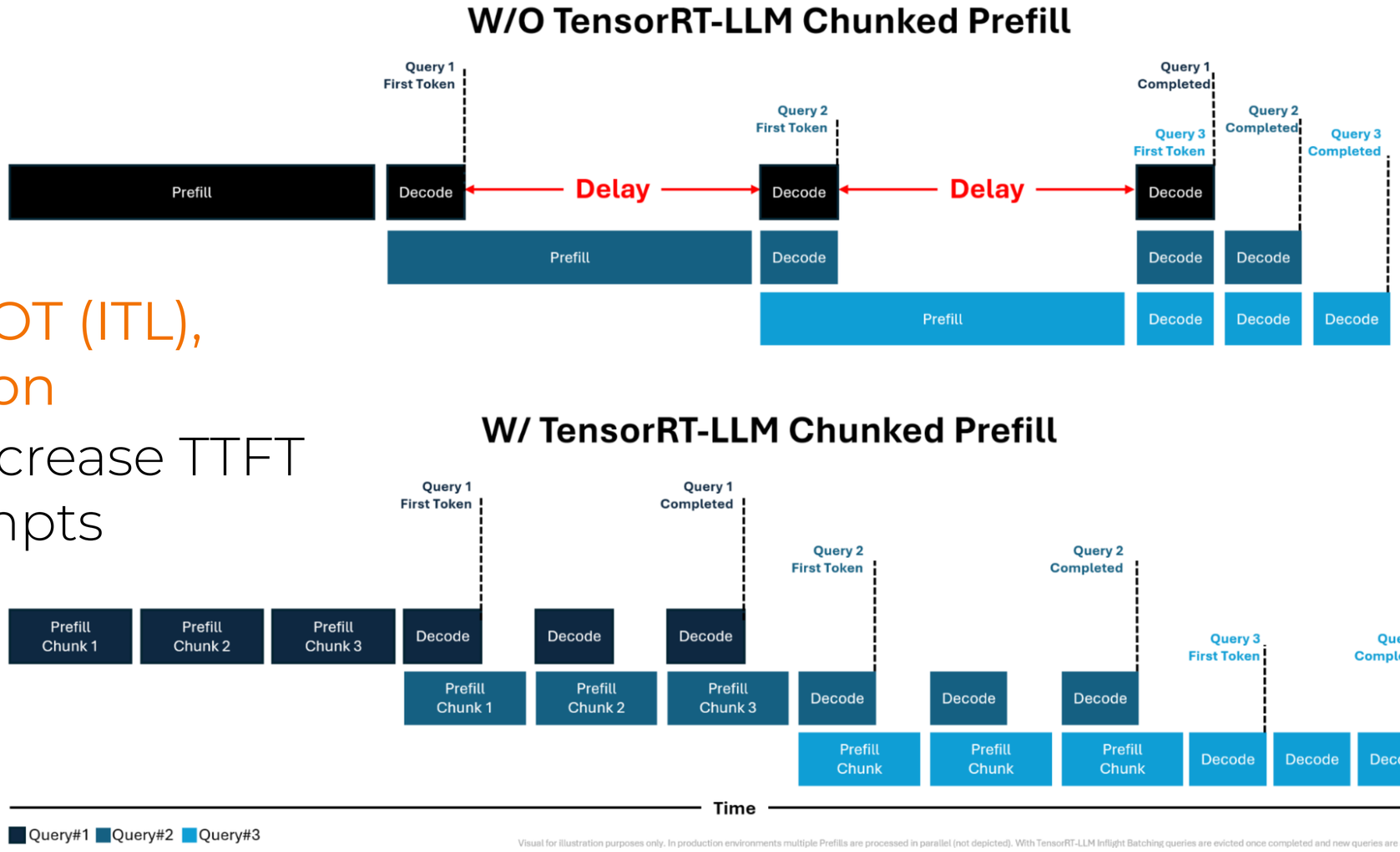
Prefill Interferes with Decode

- In vLLM documentation, "By default, vLLM scheduler prioritizes prefills and doesn’t batch prefill and decode to the same batch. This policy optimizes the TTFT (time to the first token), but incurs slower TPOT (time per output token) and inefficient GPU utilization."

- Why?
  - Prefill is a large compute-heavy GEMM
  - Decode is a sequence of small latency-sensitive steps
  - If prefill is scheduled first, current decodes stall
  - e.g. decode → decode → [large prefill] → decode
- What about slicing the prefill tokens? → chunked prefill

Chunked Prefill

Image credit: https://developer.nvidia.com/blog/streamlining-ai-inference-performance-and-deployment-with-nvidia-tensorrt-llm-chunked-prefill/



Improves TPOT (ITL), GPU utilization

But might increase TTFT for long prompts

Prefill/Decode Disaggregation

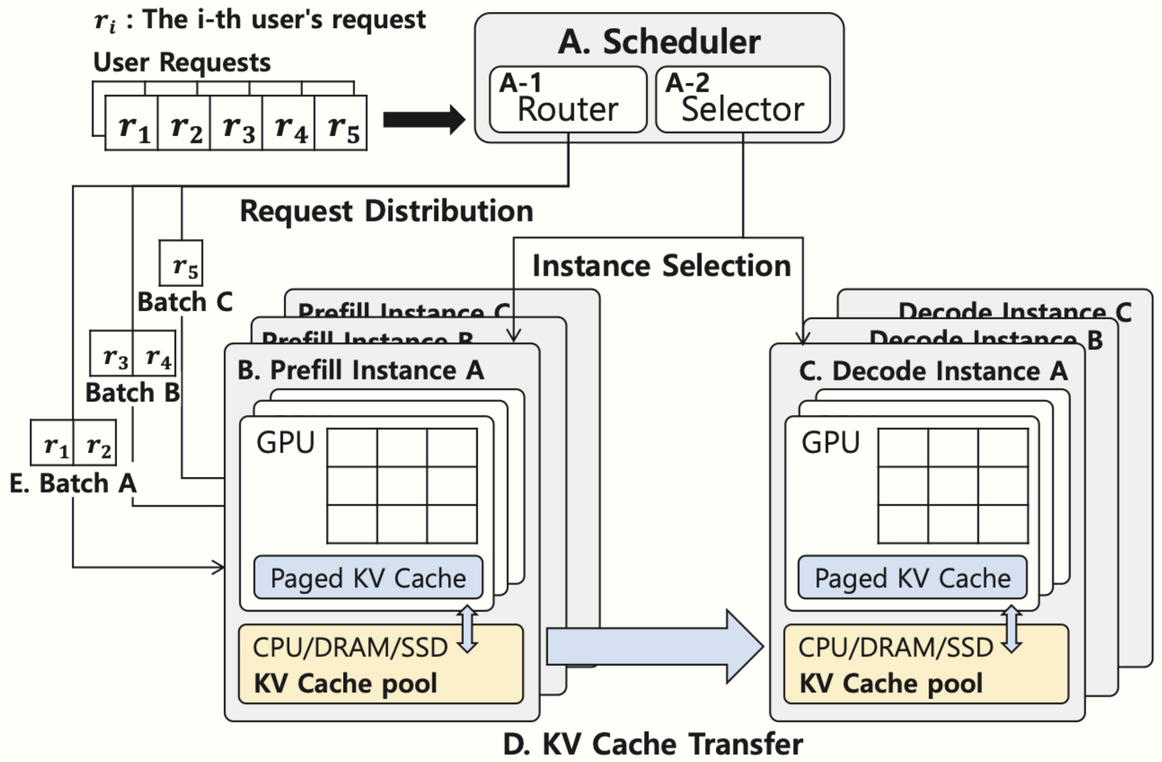
- Chunked prefill is good, however, still there are some limitations
  - Prefill and decode still share the same GPU
  - They compete for compute, memory bandwidth, etc..

  - Prioritizing prefill improves TTFT but hurts TPOT
  - Prioritizing decode improves TPOT but hurts TTFT

- Scheduling helps, but resource contention remains

Prefill/Decode Disaggregation

- Let's separate prefill and decode workers (instances)
  - Prefill instance: process input prompt and produce KV cache
  - Decode instance: load KV cache and generate tokens



Serving Systems - Speculative Decoding

LLM: Key Observations

- Autoregressive decoding generates one token at a time
- Observation 1: Some tokens are easy
  - Not all tokens are equally hard to predict
  - e.g. "What is the square root of 7? → The square root of 7 is *2.646*."

- Observation 2: Decode is memory-bound
  - So computation may not fully utilize the GPU

- Can we use small models to approximate next (and easy) tokens?

Speculative Decoding [Leviathan+ 2022]

- Speculative decoding exploits both observations
- Observation 1: Some tokens are easy
  - → A small draft model can guess them

- Observation 2: Decode is memory-bound
  - → The target model can verify several tokens in parallel

Sp

- S
- C
  -
- C
  -

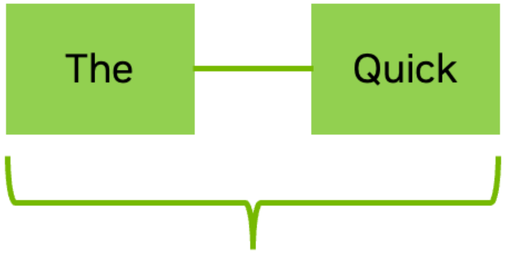


Image credit: https://developer.nvidia.com/blog/an-introduction-to-speculative-decoding-for-reducing-latency-in-ai-inference/

Sp

- S
- C
  -
- C
  -

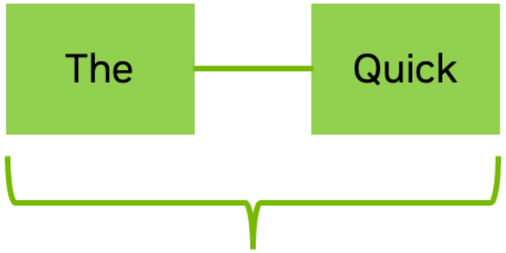


Image credit: https://developer.nvidia.com/blog/an-introduction-to-speculative-decoding-for-reducing-latency-in-ai-inference/

Speculative Decoding [Leviathan+ 2022]

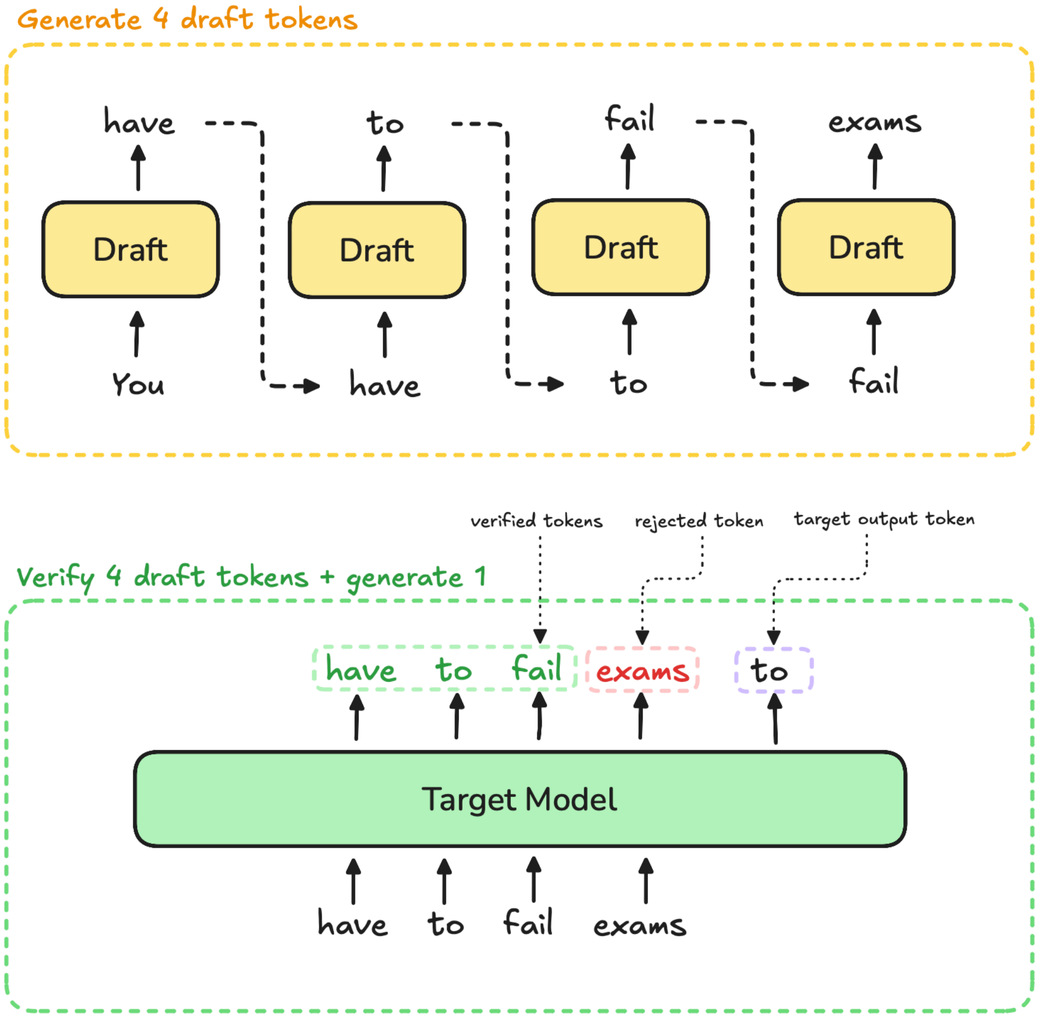
- Draft model q: small and fast model, proposes tokens
- Target model p: large and accurate model, verifies tokens

- Draft model generates candidate tokens
- Target model checks whether those tokens are consistent with p
- Final output should match the target model distribution

- Important property: the draft model speeds up decoding,
  but should not change the output distribution

Speculative Decoding [Leviathan+ 2022]

- Draft-and-verify
  - Drafting is sequential, but verification of a given sequence is parallel
  - The target model is not generating all tokens at once
  - It evaluate the likelihood of the draft tokens with a single pass



Speculative Decoding [Leviathan+ 2022]

- Algorithm
  - Given prefix x, q samples γ tokens $y_1, \ldots, y_\gamma \sim q$
  - Target model computes $p(y_i \mid x, y_{<i})$

  - For all draft tokens in one forward pass
    - Accept draft tokens from left to right
    - Stop at the first rejection

    - If all draft tokens are accepted, sample one extra token from p

Speculative Decoding [Leviathan+ 2022]

- Acceptance rule
  - For each draft token y_i, calculate $\alpha_i = \min\left(1, \frac{p_i(y_i)}{q_i(y_i)}\right)$
    - If target model likes the token more than draft model ($\alpha_i \geq 1$):
      - accept always
    - If draft model overestimated the token:
      - reject with probability $1 - \frac{p_i(y_i)}{q_i(y_i)}$

- e.g. $p_i(y_i) = 0.9,\ q_i(y_i) = 0.4$ → accept always
- e.g. $p_i(y_i) = 0.12,\ q_i(y_i) = 0.20$ → reject with 40% prob.

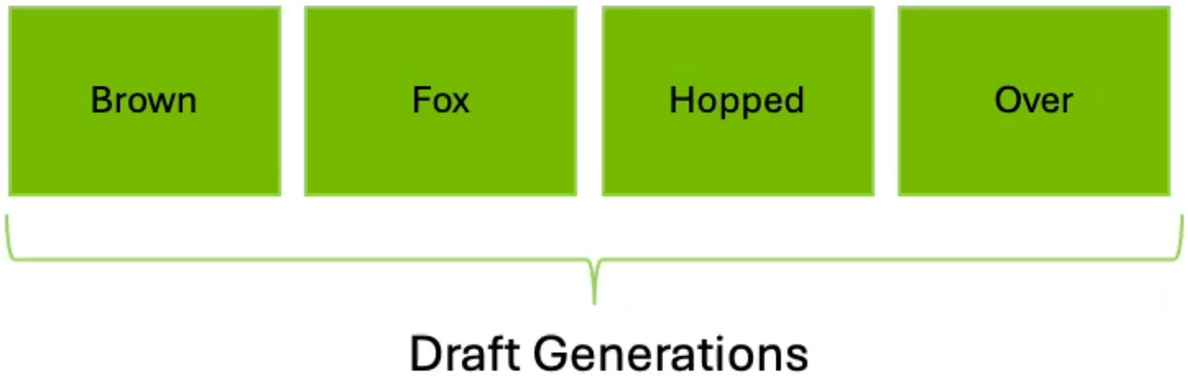


Image credit: https://developer.nvidia.com/blog/an-introduction-to-speculative-decoding-for-reducing-latency-in-ai-inference/

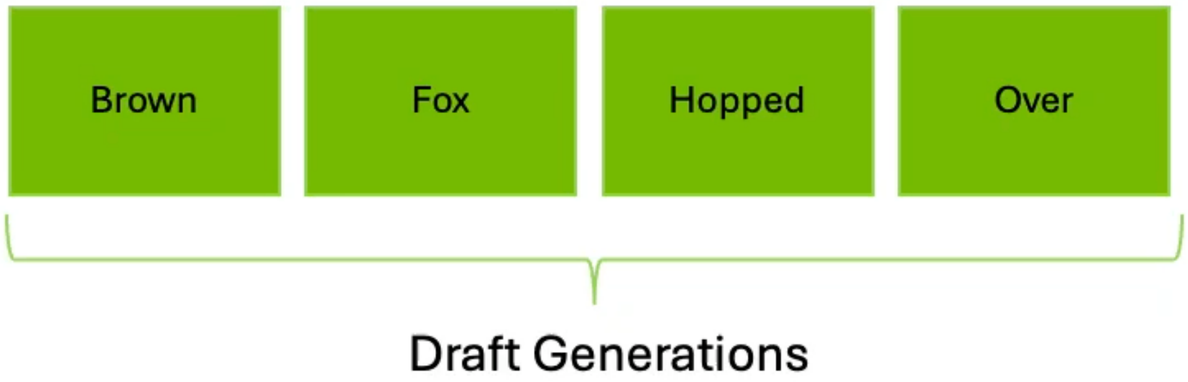


Image credit: https://developer.nvidia.com/blog/an-introduction-to-speculative-decoding-for-reducing-latency-in-ai-inference/

Speculative Decoding [Leviathan+ 2022]

- Rejection sampling
  - If a draft token is rejected:
    - Do not simply sample from distribution p_i
    - Rather, sample from the residual distribution

$$r_i(v) = \frac{\max(p_i(v) - q_i(v), 0)}{\sum_u \max(p_i(u) - q_i(u), 0)}$$

  - Why?
    - Accepted tokens already used probability mass $\min(p, q)$
    - Rejection fills the leftover mass $[p - q]_+$

Speculative Decoding [Leviathan+ 2022]

- Example:
  - 1. Draft model: $q(A) = 0.2$, $q(B) = 0.5$, $q(C) = 0.3$
  - 2. Target model: $p(A) = 0.5$, $p(B) = 0.3$, $p(C) = 0.2$
  - 3. Accept prob: $\alpha(A) = 1.0$ $\alpha(B) = 0.6$ $\alpha(C) = 0.667$
    - If draft samples A: always accept, B: 60% accept, C: 66.7% accept

  - If rejection happens,
    - $\max(p(v) - q(v), 0)$ are A: 0.3, B = 0.0, C = 0.0
    - Hence, rejection sampling always choose A

Image credit: https://research.google/blog/looking-back-at-speculative-decoding/

Speculative Decoding [Leviathan+ 2022]

Image credit: https://research.google/blog/looking-back-at-speculative-decoding/

Speculative Decoding [Leviathan+ 2022]

Speculative Decoding [Leviathan+ 2022]

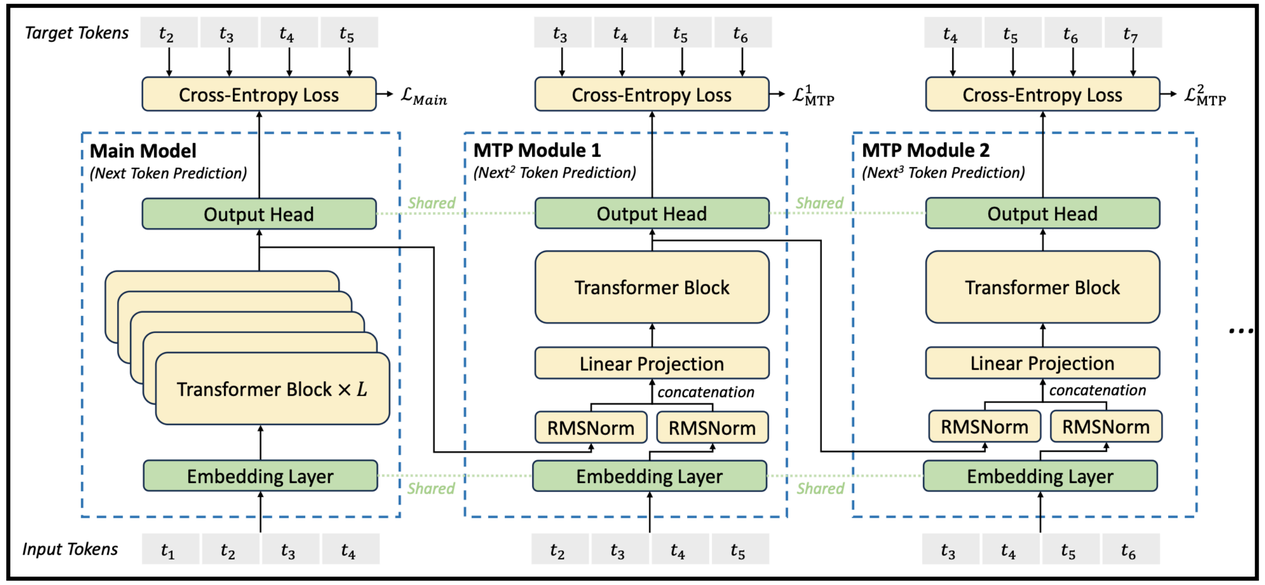
- Which draft model should we choose?
  - Speculative decoding is useful only if the draft is,
    - fast enough to be cheap
    - accurate enough to get high acceptance
    - compatible with the target model and serving stack

- In practice,
  - Use small and same-family draft model if available
    - e.g. target 70B, draft 7B or 13B

Speculative Decoding [Leviathan+ 2022]

- Which draft model should we choose?
  - Speculative decoding is useful only if the draft is,
    - fast enough to be cheap
    - accurate enough to get high acceptance
    - compatible with the target model and serving stack

- In practice,
  - If model support, we can consider MTP,
    avoids hosting another draft model

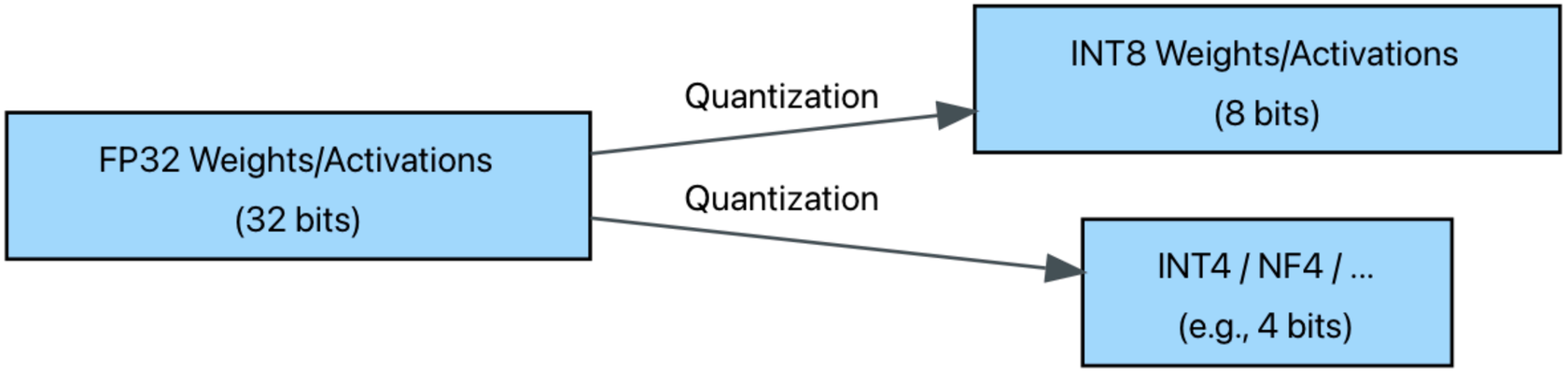


Serving Systems - Quantization

Quantization

- Key idea: reduce the precision of numbers

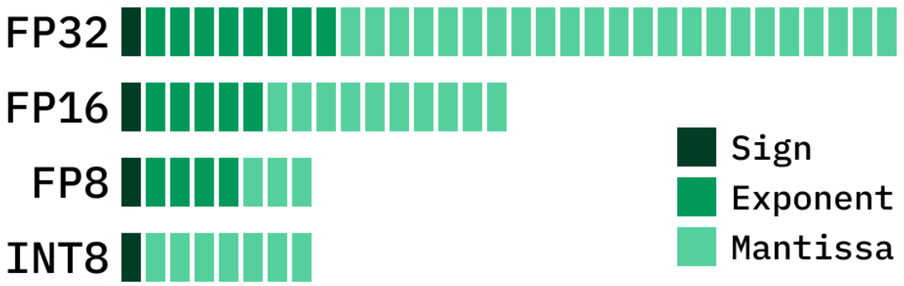
- Decode is memory-bound → fewer bytes means proportional speedup
- BF16 → INT4 weights: theoretical 4× speedup; KV is separate
- Also saves GPU memory → larger batch → additional throughput gain
- Unlike training, inference is far more tolerant of quantization



Quantization

- fp32 (4 bytes): needed for params. and optimizer states during training
- bf16 (2 bytes): default for inference

- fp8 (1 byte) [-240, 240] for e4m3 on H100s
- int8 (1 byte) [-128, 127]
  - Less accurate but cheaper, but for inference only [Baalen+ 2023]
- int4 (0.5 bytes) [-8, 7]
  - Cheaper, even less accurate [Baalen+ 2023]



What Do We Quantize?

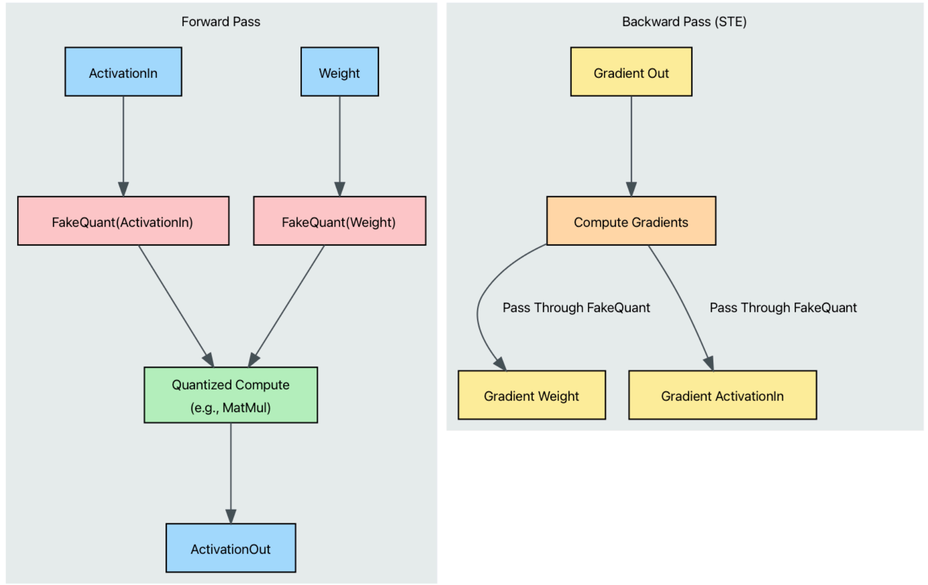
- Weight-only quantization (W4A16, etc.)
  - Most common form, directly accelerates decode
  - Representatives: GPTQ, AWQ

- KV-cache quantization
  - Eases the KV memory pressure itself → larger batch, longer context
  - FP8 KV is effectively the standard on H100/H200

- Activation quantization (W8A8, W4A8)
  - Accelerates prefill compute as well
  - Representatives: SmoothQuant, ZeroQuant

Weight-Only Quantization

- Quantization-aware training (QAT)
  - During training, quantize-and-dequantize (fake quantization) during the forward pass to simulate quantization errors
  - 👍: weights are trained to work with quantization
  - 👎: requires expensive large-scale training



Weight-Only Quantization

- Post-training quantization (PTQ)
  - Done after training, so much cheaper
  - Run on sample data to determine scale and zero point for each layer or tensor

  - GPTQ [Frantar+ 2022]
    - Use Hessian information to update non-quantized weights to account for quantization error

vLLM

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

Easy, fast, and cheap LLM serving for everyone

| Documentation | Blog | Paper | Twitter/X | User Forum | Developer Slack |

🔥 We have built a vLLM website to help you get started with vLLM. Please visit vllm.ai to learn more. For events, please visit vllm.ai/events to join us.

vLLM

vLLM is fast with:

- State-of-the-art serving throughput
- Efficient management of attention key and value memory with **PagedAttention**
- Continuous batching of incoming requests, chunked prefill, prefix caching
- Fast and flexible model execution with piecewise and full CUDA/HIP graphs
- Quantization: FP8, MXFP8/MXFP4, NVFP4, INT8, INT4, GPTQ/AWQ, GGUF, compressed-tensors, ModelOpt, TorchAO, and more
- Optimized attention kernels including FlashAttention, FlashInfer, TRTLLM-GEN, FlashMLA, and Triton
- Optimized GEMM/MoE kernels for various precisions using CUTLASS, TRTLLM-GEN, CuTeDSL
- Speculative decoding including n-gram, suffix, EAGLE, DFlash
- Automatic kernel generation and graph-level transformations using torch.compile
- Disaggregated prefill, decode, and encode

Summary

Technique	TTFT	TPOT/ITL	Throughput	Goodput	$/tokens
Continuous batching	+	+	++	++	+
PagedAttention	+	+	++	++	++
Prefix caching	++	0/+	+	+	+
Chunked prefill	±	++	+	++	+
P/D disaggregation	++	++	+/++	++	±/+
Speculative decoding	0/+	++	++	++	+/++
Quantization	+	+/++	+/++	+	++

Summary

- Decode is memory-bound: Most inference technique derives from this
- KV cache is the system's #1 pressure
- Algorithm ↔ system ↔ model compression are three orthogonal axes
  - Apply them together for impact
- TTFT/TPOT/Goodput/$ each demand different techniques