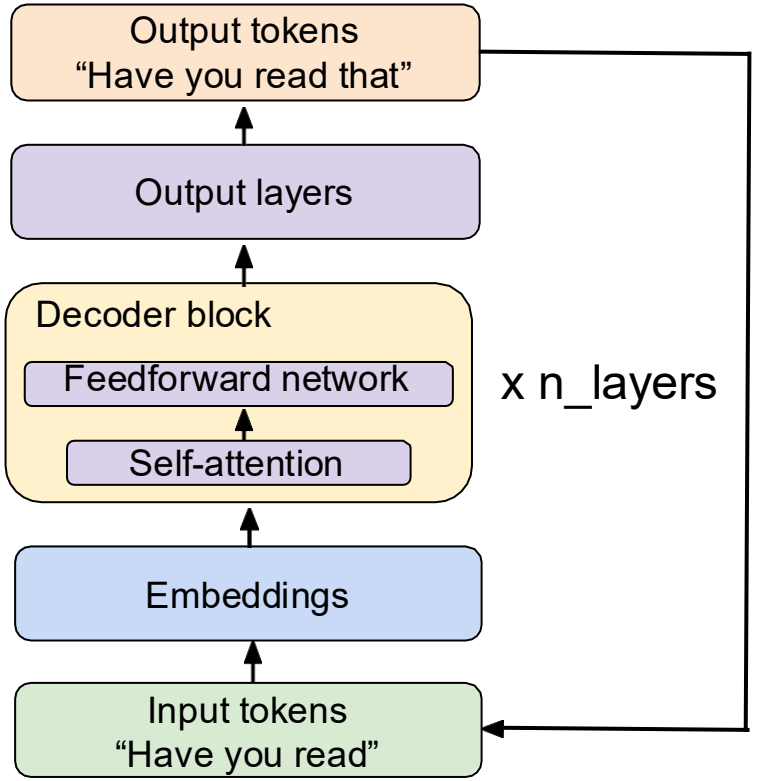
Large Language Model Distributed Inference

March 3rd, 2026

Llama Models

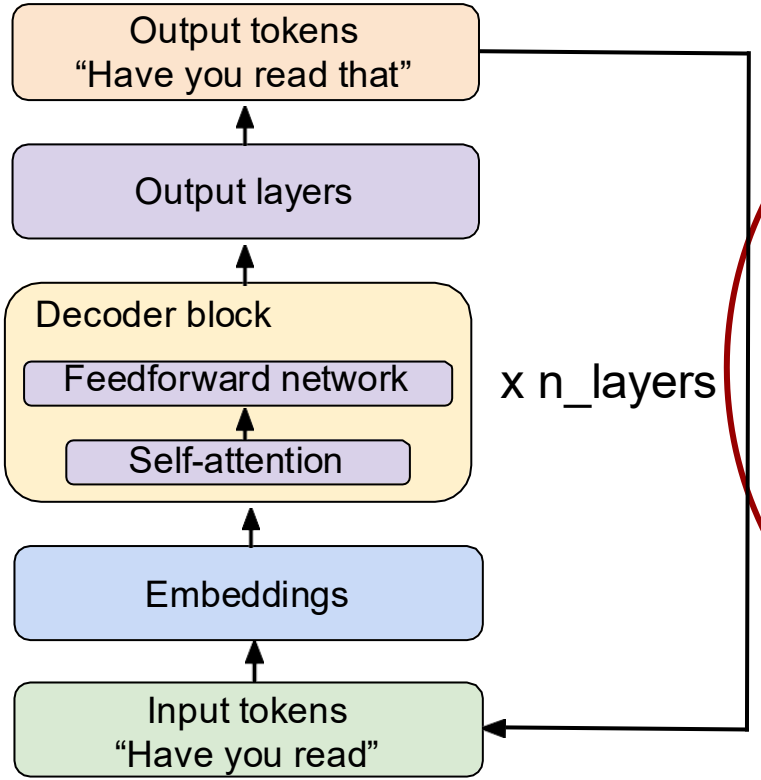
- Family of decoder-only transformer large language models
- Trained model weights released by Meta AI starting in February 2023
- Latest generation released in July 2024
  - Llama 3 models with 8B, 70B, or 405B parameters



Training vs Inference in Llama models

Training

Process of learning model weights by optimizing performance on tasks such as next-token prediction.



Inference

Process of using trained model weights to provide outputs based on given input prompts.

Exercise: Assessing Memory Needs

1) Compute the memory that the model weights of the Llama model with 405 billion weights will take. Assume weights are float16.

2) Will this fit on one GPU? If not, how many H100 GPUs would you need?

GPUs:

H100 ⇒ 80 G
A100 ⇒ 40 G

Large Memory Needs for Large Models

Memory Needed for Model Weights only

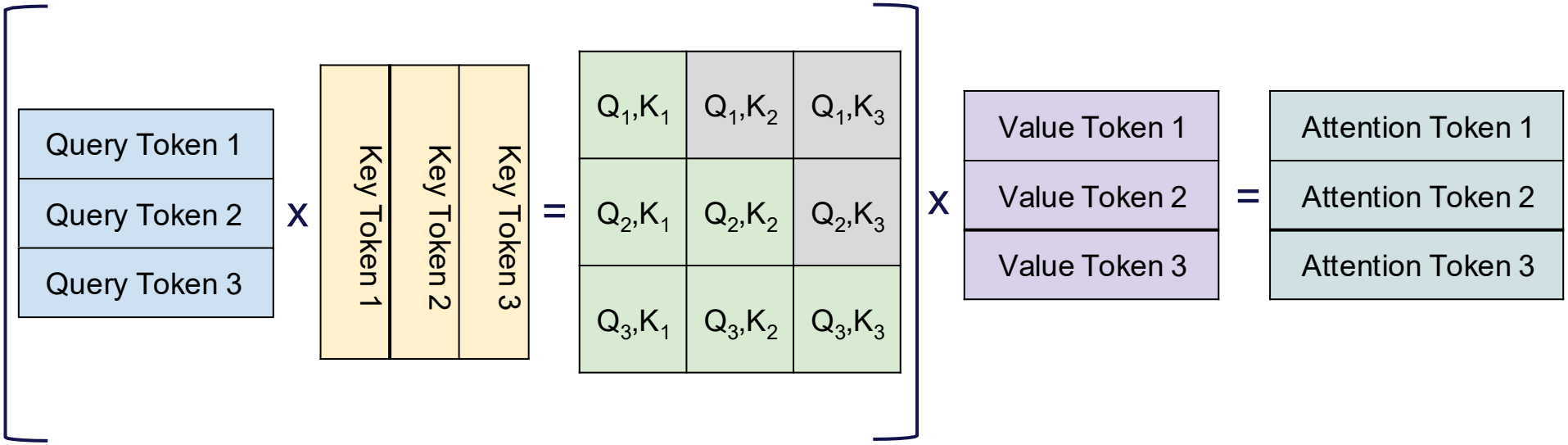
Model Size	Float16	Float8	INT4
8B	16 GB	8 GB	4 GB
70B	140 GB	70 GB	35 GB
405B	810 GB	405 GB	203 GB

Adapted from https://huggingface.co/blog/llama31

KV Caching

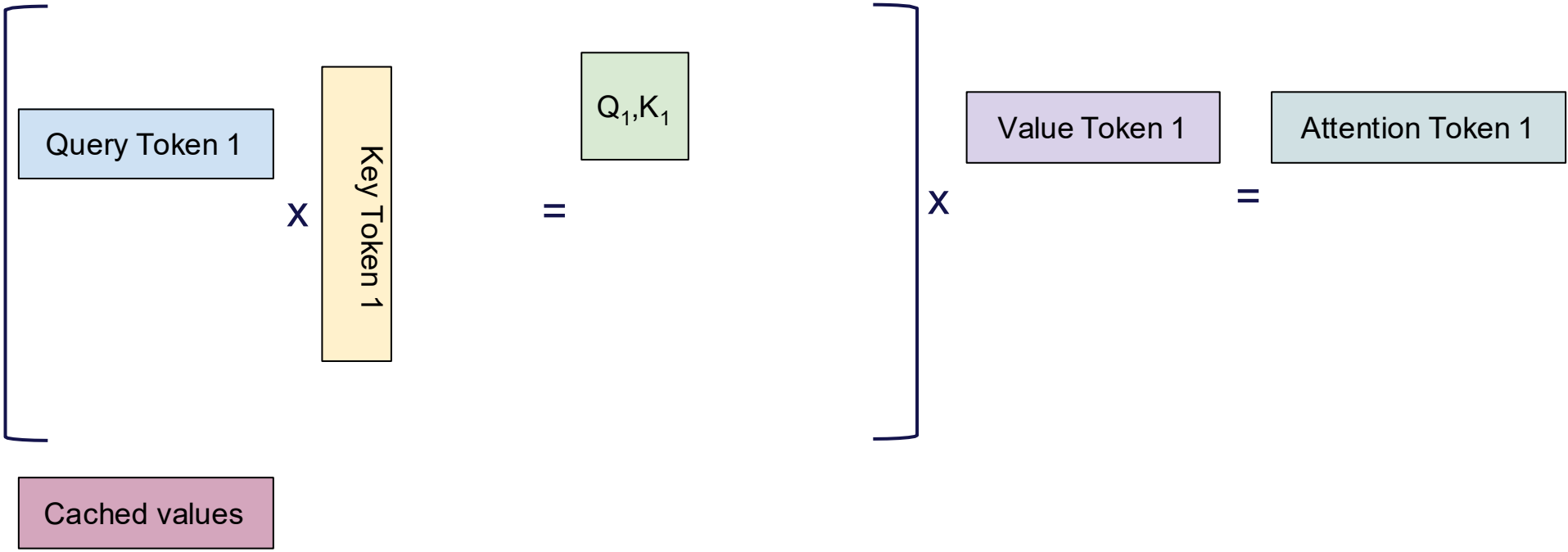
- Storing key and values for **previous tokens** to speed up inference
- Used in decoder architectures when generating multiple tokens

Self-Attention

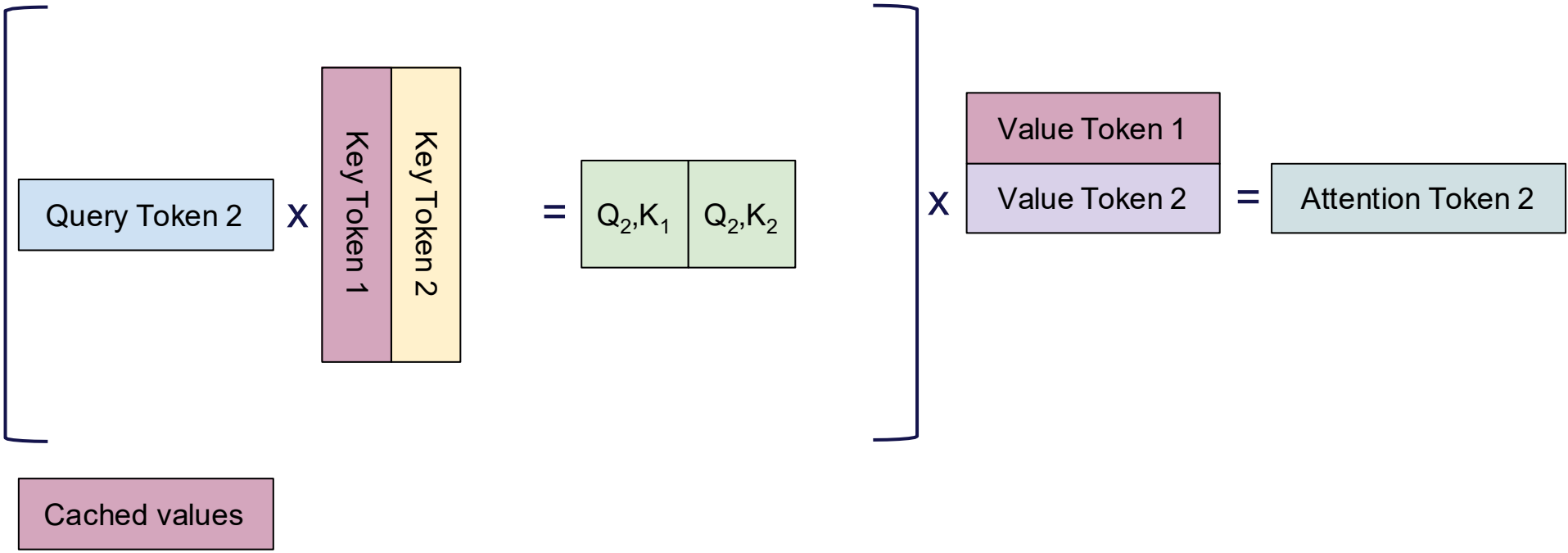


We keep re-computing earlier key and value tokens!

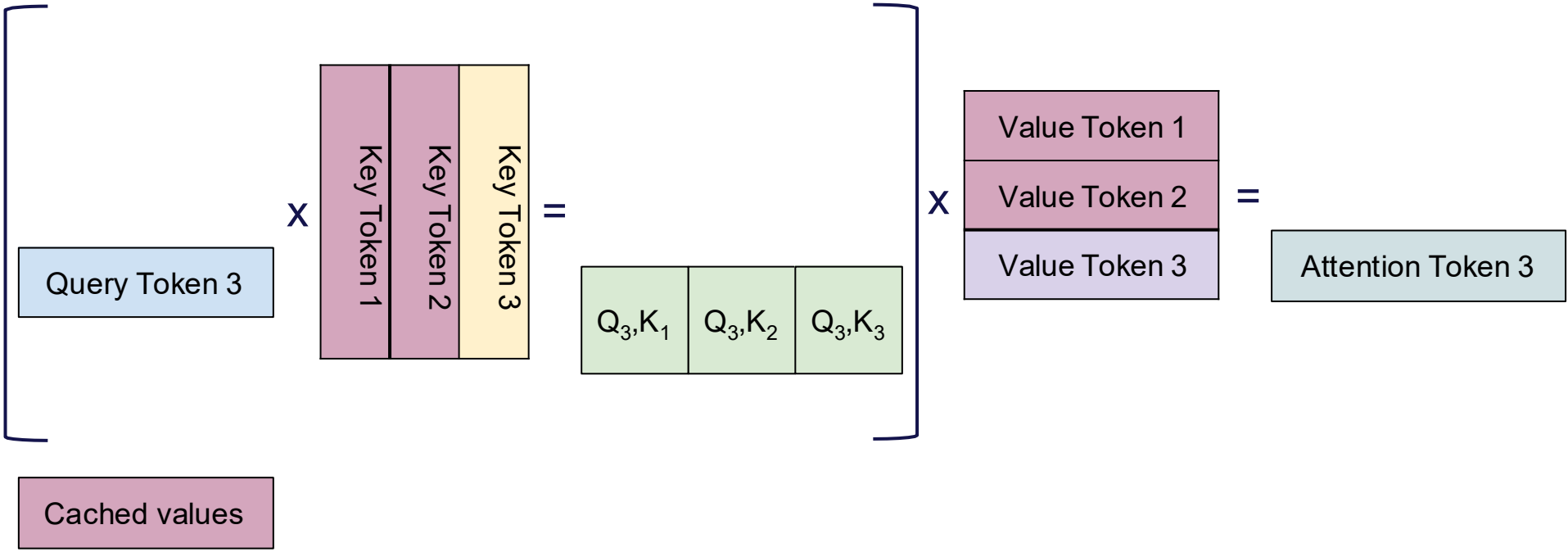
Self-Attention with KV-Caching



Self-Attention with KV-Caching



Self-Attention with KV-Caching



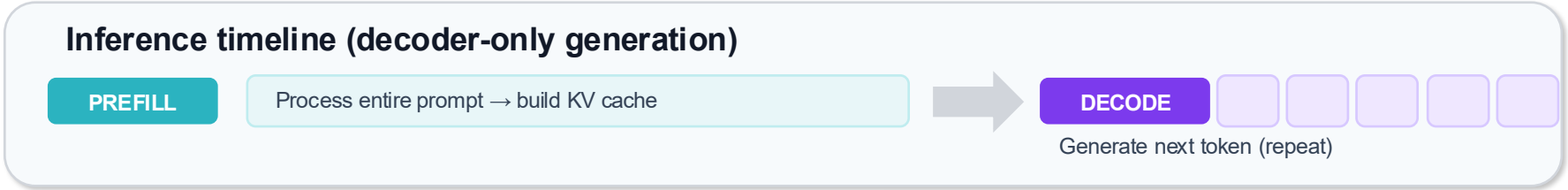
Adapted from https://medium.com/@joaolages/kv-caching-explained-276520203249

KV Caching

- Storing key and values for previous tokens to speed up inference
- Used in decoder architectures when generating multiple tokens

- Pro: small matrices = faster matrix multiplication = faster inference
- Con: need more GPU memory to cache key and value states

Putting together: Inference Pipeline



Prefill: throughput-oriented

- Runs a full forward pass on the prompt tokens
- Heavily parallel (matrix multiplications across tokens)
- Main output: KV cache for each layer (plus prompt logits)
- Strongly impacts TTFT (time-to-first-token)

Decode: latency-oriented

- Autoregressive loop: one token per step (depends on previous tokens)
- Uses KV cache so attention reuses past keys/values
- Per-token work grows with context length (memory traffic matters)
- Main metric: tokens/sec (and tail latency)

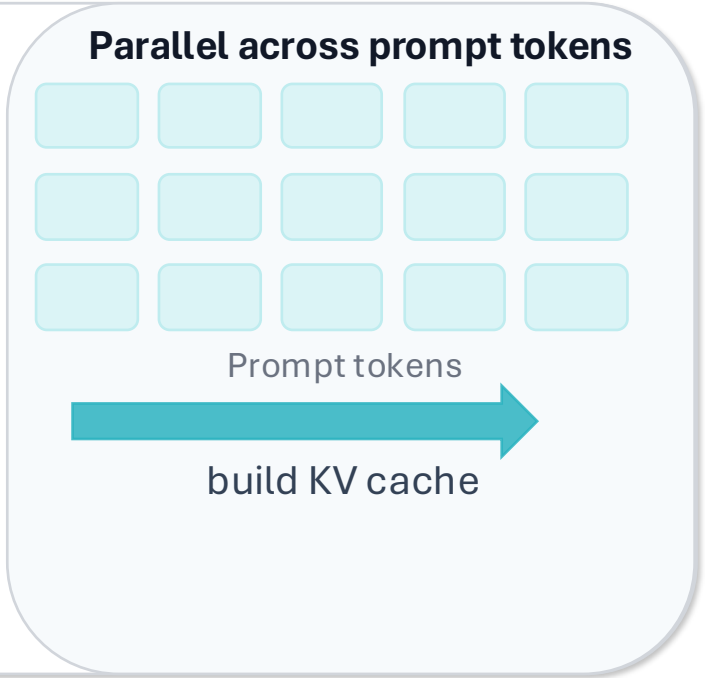
1. Prefill: process the prompt in parallel

What happens

- Tokenize prompt → embeddings
- Run N decoder layers on all prompt tokens
- Materialize KV cache for every layer (K/V per token)
- Compute logits for the next token (TTFT ends here)

Performance profile

- High arithmetic intensity (GEMMs), good GPU utilization
- Cost scales with prompt length (and batch size)
- Usually easier to batch than decode (no step-to-step dependency)



2. Decode: one token at a time (autoreg loop)

Per-token decode step

repeat until stop token

1) Take previous token(s) + position

2) Compute Q for the new token (per layer)

3) Attention: Q × (cached K/V) → context

4) MLP + output logits → sample/argmax

5) Append new K/V to the cache

What dominates decode time?

- Dependency chain: step k needs token k−1 → limited parallelism
- Reading large KV cache (bandwidth/latency), especially for long context
- Small kernels & launch overhead can matter at low batch size
- Sampling / beam search adds work and can increase variance

Decode optimization often looks like: better attention kernels + better cache management + smarter scheduling.

Inference Optimizations

Prefill-focused

- Fused / IO-aware attention kernels (FlashAttention)
- Larger batch sizes (within TTFT SLO)
- Weight quantization (FP16 → INT8/INT4) to fit bigger batches
- Prompt caching (reuse KV for repeated prefixes)

Decode-focused

- Efficient KV cache allocation (PagedAttention / paging)
- Continuous batching (interleave decode steps across requests)
- Speculative decoding (draft + verify multiple tokens per step)
- KV cache quantization / compression (memory savings)

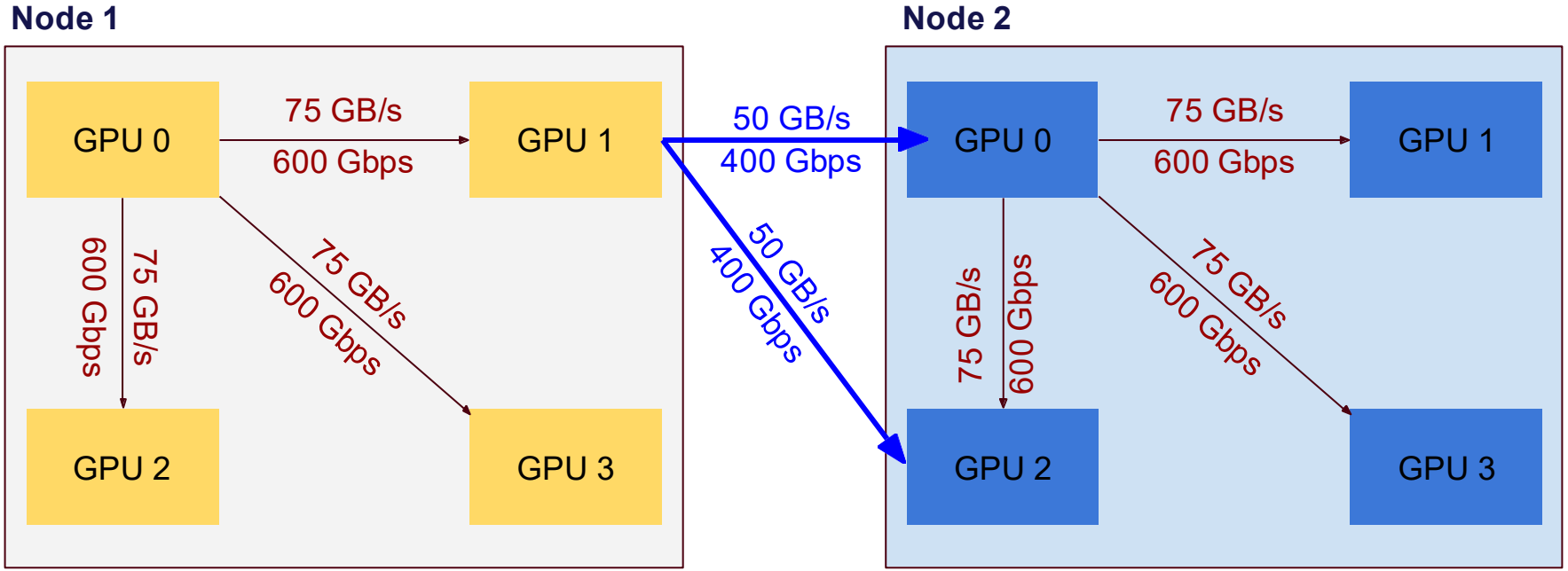
Tuning mindset: for chatbots, TTFT is often the first SLO → optimize prefill.
For long completions & high QPS, decode throughput and KV cache efficiency dominate.

Large Memory Needs for Large Models

Model Size	Memory for Float16 Weights	Memory for KV Cache for 128k token request	Number of H100s needed for both
8B	16 GB	15.62 GB	1
70B	140 GB	39.06 GB	2.24
405B	810 GB	123.05 GB	11.66

Adapted from https://huggingface.co/blog/llama31

GPU-to-GPU Communication

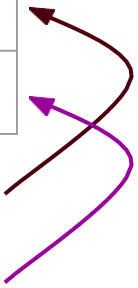


Inside Node (**NVLINK**): Each GPU talks to other three GPUs at 75 GB/s (single direction). This sums up to 900 GB/s all GPU-GPU bidirectional speed. 75 GB/s * 6 * 2 = 900 GB/s

Outside Node (**InfiniBand Network NDR**): Each GPU communicates to other GPUs in another node at 400 Gbps (50 GB/s).

Large Memory Needs for Large Models

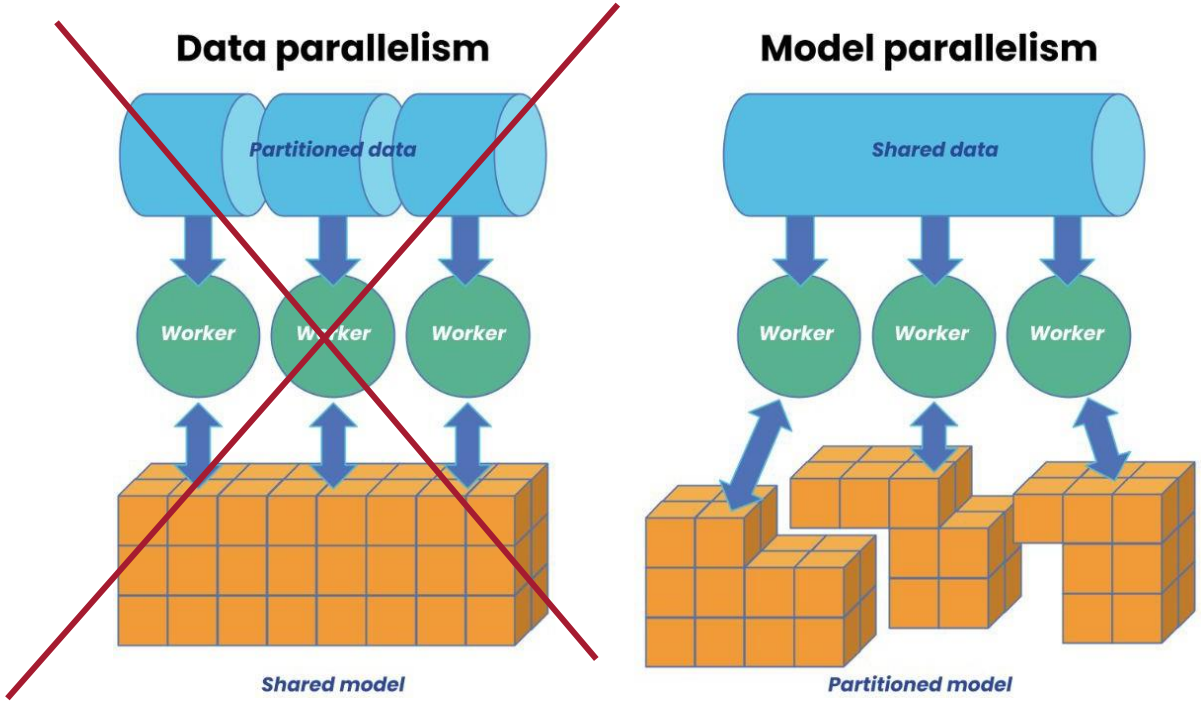
Model Size	Memory for Float16 Weights	Memory for KV Cache for 128k token request	Number of H100s needed for both
8B	16 GB	15.62 GB	1
70B	140 GB	39.06 GB	2.24
405B	810 GB	123.05 GB	11.66



Virtual Large Language Model (vLLM)

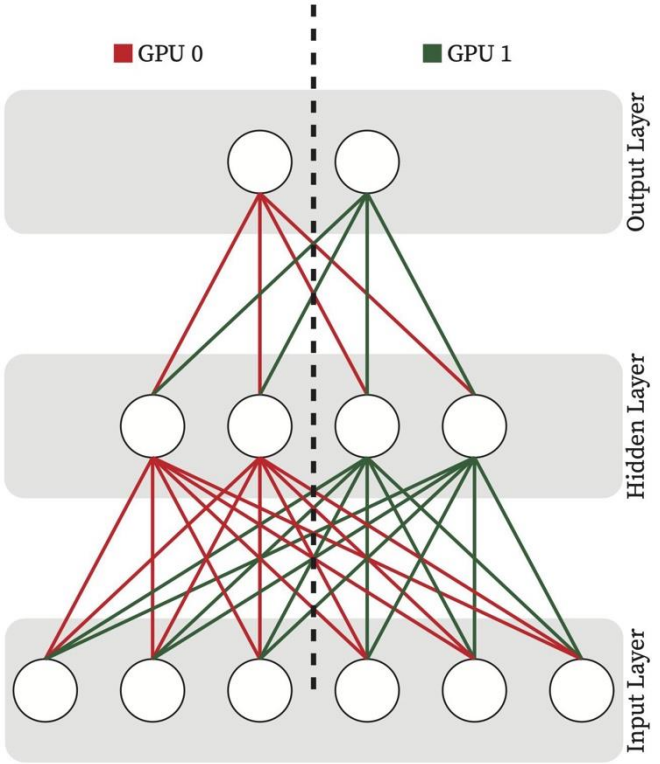
- Open-source library that supports fast and efficient inference and serving of LLMs
- Has built-in support for distributed inference through tensor and pipeline parallelism
- We'll use it with Llama models but supports many other models (will discuss this further later)

Types of Parallelism



https://www.anyscale.com/blog/what-is-distributed-training

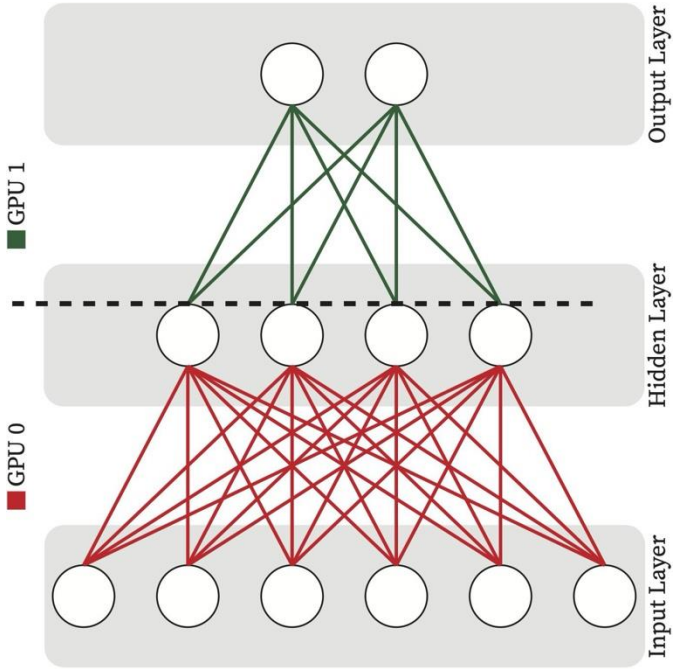
Single-Node Multi-GPU: Tensor Parallel Inference



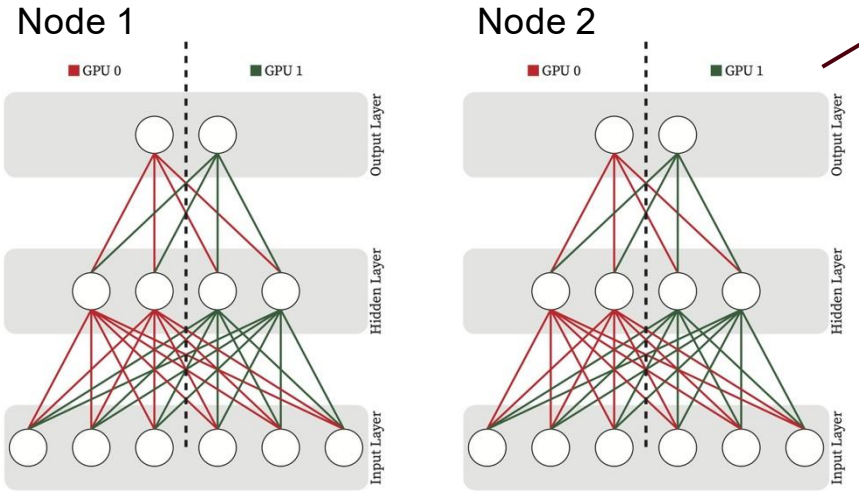
- All GPUs contribute in each layer computation
- Removes the GPU Idle time

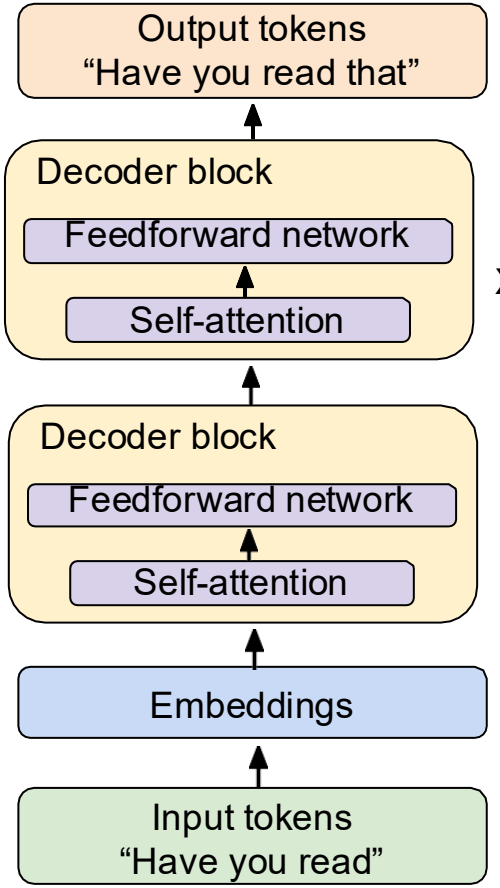
https://docs.vllm.ai/en/latest/serving/distributed_serving.html

Pipeline Parallel Inference



Multi-Node Multi-GPU: Tensor Parallel & Pipeline Parallel Inference





Behind the Scenes of vLLM Distributed Inference

- vLLM uses Megatron-LM's tensor parallel algorithm

- Manages distributed multi-node inference scheduling using Ray:
  - Ray is an open-source framework for scaling AI models

Thank you