Serving and Training a Transformer

David I. Inouye

Dynamic Content · Last Updated: November 2, 2026

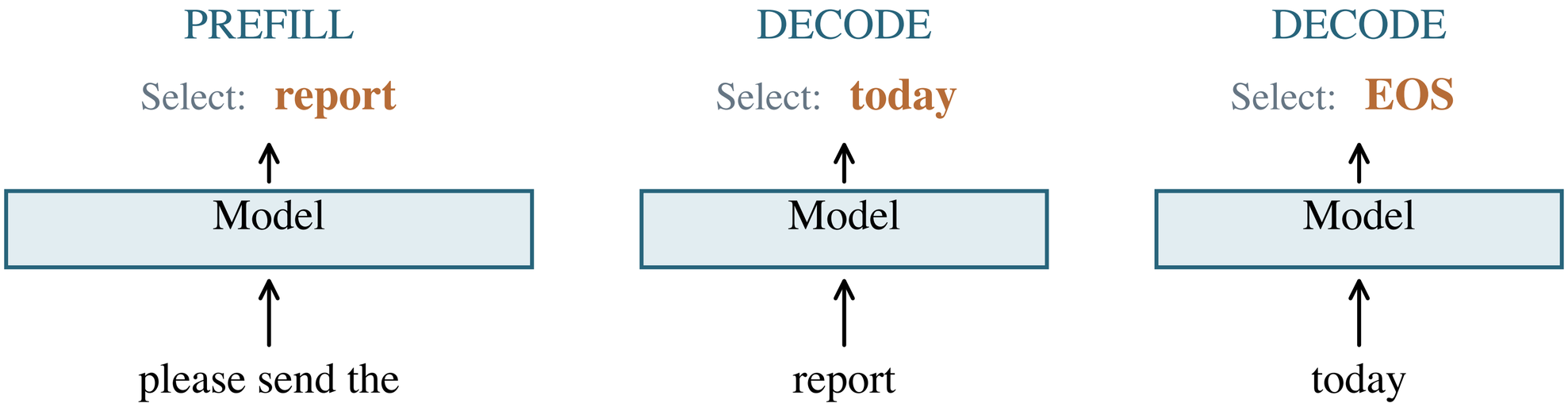
Serving a model means managing user latency and request memory

Prefill and decode → why K/V can be reused → memory → context growth.

Then: data movement, throughput and inference diagnosis.

First serve the trained model; afterward, consider training it.

Prefill reads the prompt; decode extends it one token at a time

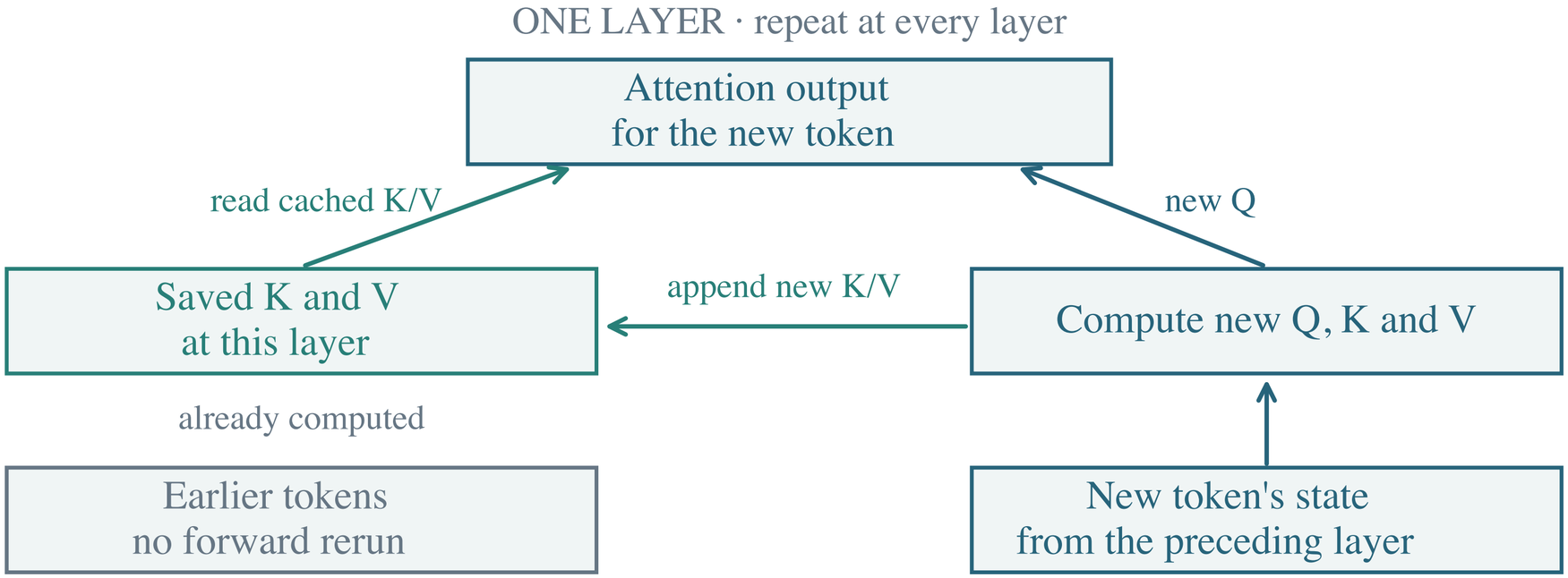


Sequence of passes: prefill → decode → decode (synthetic tokens, not timing)

Work	Inputs already known	User-facing measure
Prefill	All prompt tokens	Time to first token
Decode	One new input token per step	Time per subsequent token

Prefill saves per-layer keys and values; each decode step reuses and extends them.

Earlier tokens stay fixed; attention needs only their keys and values



- **Causal masking:** appending a token cannot change earlier token states.
- **Attention mixes tokens;** MLPs, normalization and residual paths act within each token.

Keep old K/V at every layer; discard old queries and other intermediate states.

Inference keeps weights and K/V, plus temporary workspace

Memory term	Why it stays or grows
Model weights	Shared across requests; size depends on parameters and precision
K/V at every layer	Retained for later tokens; grows with prefix length and active requests
Temporary activations/workspace	Needed for current computations; reused or freed afterward

We cache the history attention will read—not every activation produced for that history.

A longer context grows both the cache and the attention work

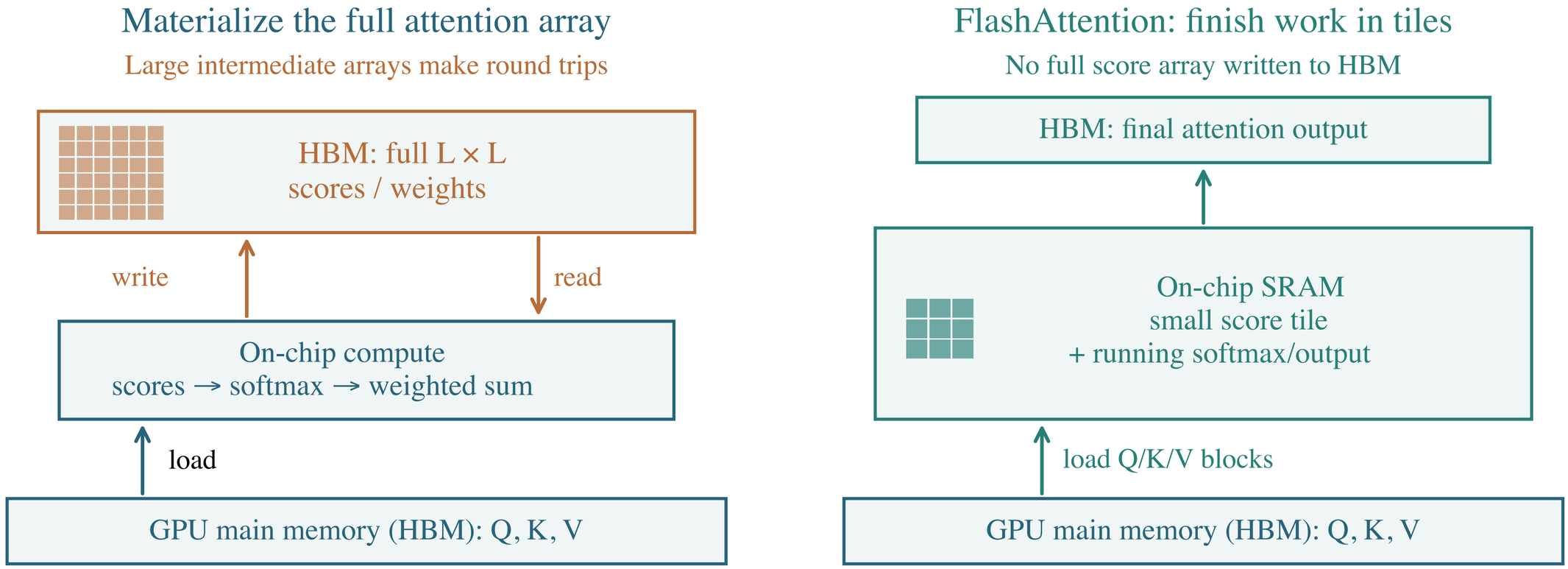
Quantity being counted	If context length doubles: $L \to 2L$
Compute: dense attention over a full prompt	About **4×** as many query–key pairs
Compute: cached attention for one new token	About **2×** as many query–key pairs
Memory: persistent K/V cache	**2×** as many stored K/V entries

Fixed model, precision and request count. **Operation counts are not measured latency.**

Conventional multi-head cache bytes $\approx 2NBLdb$.

Synthetic example: $N = 12$ layers, $B = 1$, $L = 1024$, $d = 512$, $b = 2$ bytes → **24 MiB**.

Fewer memory transfers can make the same attention faster



Same dense attention; fewer large memory round trips.

Tiling reduces intermediate storage and traffic; the pairwise arithmetic remains quadratic.

Higher throughput need not mean a faster response for each user

Measure	Main dependencies—and why
Time to first token	Queue/load + prompt length: wait for service, then prefill the prompt
Time per subsequent token	Model size + context + active batch: each step reads weights/K/V and computes the new token
Aggregate output tokens/sec	Batch size + hardware utilization: process several requests together to share weight access and parallel work

Larger batches can raise throughput while adding queueing, cache memory and per-step work.

More output tokens mean more sequential decode steps—and a longer total response time.

Which memory grows when the prompt grows?

> **Fixed weights and request count; cached inference now runs out of memory on longer prompts.**
> **What should we inspect first?**
> **A. Parameter gradients and optimizer state**
> **B. KV cache and temporary activations**
> **C. The number of learned weights**

B. Prefix-dependent memory grows even though the model's weights do not.

Compare **task quality, user latency, throughput and peak memory** before choosing a fix.

Training adds backward computation and parameter updates

Memory ledger → precision and recomputation → sharing state across GPUs.

Serving keeps weights fixed. Training must also compute gradients and update them.

Full fine-tuning can need far more memory than serving

Qwen3-8B: approximately 8.2 billion parameters. Illustrative full-parameter AdamW setup:

Stored arrays	Bytes / parameter	Approximate memory
16-bit weights used for computation	2	16.4 GB
FP32 master weights for updates	4	32.8 GB
FP32 gradients	4	32.8 GB
Two FP32 Adam moment arrays	8	65.6 GB
Training state subtotal	**18**	**147.6 GB**

Inference: 16.4 GB weights + K/V + workspace.

Training: 147.6 GB state + saved activations + workspace.

Training can save memory by changing precision or recomputing activations

Choice	What changes	What to check
Mixed precision	Use lower precision for suitable operations; retain higher precision where needed	Numerical stability, quality and runtime
Activation checkpointing	Save fewer intermediates; recompute them during backward	Peak memory versus extra computation

Neither removes weights, gradients, optimizer state, and activations all at once.

Choose the technique for the memory term that is limiting the run.

ZeRO reduces duplicated training state across GPUs

Ordinary **data parallelism** trains on different examples with a model replica on each GPU.

ZeRO (Zero Redundancy Optimizer) partitions state among those GPUs:

Stage	State partitioned across GPUs, cumulatively
1	Optimizer state
2	Optimizer state + gradients
3	Optimizer state + gradients + parameters

Less state stored per GPU; communication supplies the pieces when needed.

Serving and training optimize different system outcomes

	Serving an LLM	Training / fine-tuning an LLM
Goal	Useful responses at acceptable latency, throughput and cost	Reach useful model quality within a compute/time budget
Memory	Weights + per-request K/V + workspace	Weights + gradients + optimizer state + saved activations
Levers	Cache K/V; manage context and batching; reduce attention traffic	Mixed precision; recompute activations; shard state with ZeRO

Identify the limiting resource, change the relevant term, then recheck quality and performance.