# From Multi-Head to Multi-Head Latent Attention

*An Engineering Analysis of Attention Mechanisms*
*for Large Language Model Inference*

Trisham Patil(trishampatil@gmail.com)

July-2026

*Technical Report*

## Table of Contents

# 1. The Attention Landscape

The rapid progression of large language models has been enabled by a sequence of deliberate architectural decisions, each resolving the principal limitations of the generation that preceded it. Models such as GPT, Claude, Llama, DeepSeek, Qwen, Mistral, Gemma, and Phi represent not only advances in parameter scale but in the fundamental mechanisms by which token sequences are processed and representations are formed. At the core of each lies the Transformer architecture, introduced by Vaswani et al. (2017) in "Attention Is All You Need." Understanding why modern attention mechanisms have evolved in the direction they have requires beginning with the problem the Transformer was originally designed to solve.

## 1.1 The Limits of Sequential Models

Prior to the Transformer, the dominant paradigm for sequence modelling—spanning tasks such as machine translation, text summarisation, and language modelling—relied on Recurrent Neural Networks (RNNs) and their more capable variant, Long Short-Term Memory networks (LSTMs) (Hochreiter & Schmidhuber, 1997). These architectures processed token sequences one element at a time, maintaining a fixed-size hidden state that was updated at each step and passed forward. Information from earlier positions reached later positions only by propagating through this ordered chain of hidden state updates.

This sequential dependency imposed a fundamental computational constraint. The hidden state at position t could not be computed until the hidden state at position t−1 had been produced. This precluded parallelisation across sequence positions within a single training step, regardless of the hardware available. Modern GPUs are designed for massively parallel operations, executing thousands of arithmetic instructions simultaneously. The serial nature of recurrent computation left the majority of this capacity idle during both training and inference, with training times scaling unfavourably as sequence lengths increased.

A more subtle failure mode was the progressive attenuation of long-range dependencies during backpropagation. Gradients flowing backward through a long sequence were multiplied at each time step. Over many steps, repeated multiplications caused these gradients to either vanish toward zero or grow without bound—a phenomenon documented in detail by Bengio et al. (1994). In practical terms, this meant that a token near the beginning of a long document exerted diminishing influence on predictions made near its end, as the gradient signal connecting them decayed before it could propagate.

LSTM networks introduced gating mechanisms—input, forget, and output gates—that allowed selective retention and discarding of information across time steps. This partially mitigated gradient instability and enabled better modelling of longer-range dependencies. However, LSTMs retained the sequential computation constraint and still required all contextual information to be encoded within a vector of fixed dimension. As sequences grew longer, the network was forced to compress an increasing volume of contextual information into this bounded representation. Information was inevitably discarded, degrading the quality of representations for long documents or conversational histories.

The architectural limitation was therefore twofold: an inability to parallelise computation across sequence positions, and an inability to maintain reliable access to distant context without lossy compression into a fixed-size state vector. Both constraints became increasingly costly as researchers sought to build models capable of processing longer inputs with stronger contextual understanding.

## 1.2 Self-Attention and the Transformer Architecture

The Transformer architecture proposed a qualitatively different solution. Rather than propagating information sequentially through a chain of hidden states, self-attention allows every token in a sequence to directly interact with every other token in a single computational step. The relationship between any two positions—regardless of their distance in the sequence—is computed directly, with equal mechanism and equal cost. There is no distance penalty on long-range dependencies and no requirement that information traverse intermediate positions to reach its destination.

This architectural decision had two immediate and significant consequences for practical systems. First, the attention computation across all token pairs in a layer is independent and can be executed in parallel across the full sequence. Transformer training can therefore utilise the full parallelism available on modern GPU hardware, keeping arithmetic units at near-capacity utilisation and reducing training time substantially relative to equivalent-scale recurrent models. Second, because every token attends to every other token directly, the model can represent dependencies across the full context window without attenuation. A token at position 1 and a token at position 4,096 interact through exactly the same mechanism as adjacent tokens, with no loss of fidelity attributable to distance.

The decoder-only variant of the Transformer—which generates text autoregressively by conditioning each new token on all previously generated tokens—became the dominant architecture for large language modelling. GPT (Brown et al., 2020) demonstrated that large-scale pretraining of decoder-only Transformers produced models with strong few-shot generalisation capabilities. Subsequent model families—including Llama (Touvron et al., 2023), Mistral (Jiang et al., 2023), Gemma (Team et al., 2024), Phi (Abdin et al., 2024), Qwen (Qwen Team, 2024), and DeepSeek (DeepSeek-AI, 2024)—each refined the Transformer in specific ways, adjusting normalisation schemes, positional encoding strategies, feedforward architectures, and activation functions. Claude (Anthropic, 2024) similarly builds upon this foundation. Despite these variations, self-attention over a full context window remains the shared computational primitive across virtually all competitive large language models in production today.

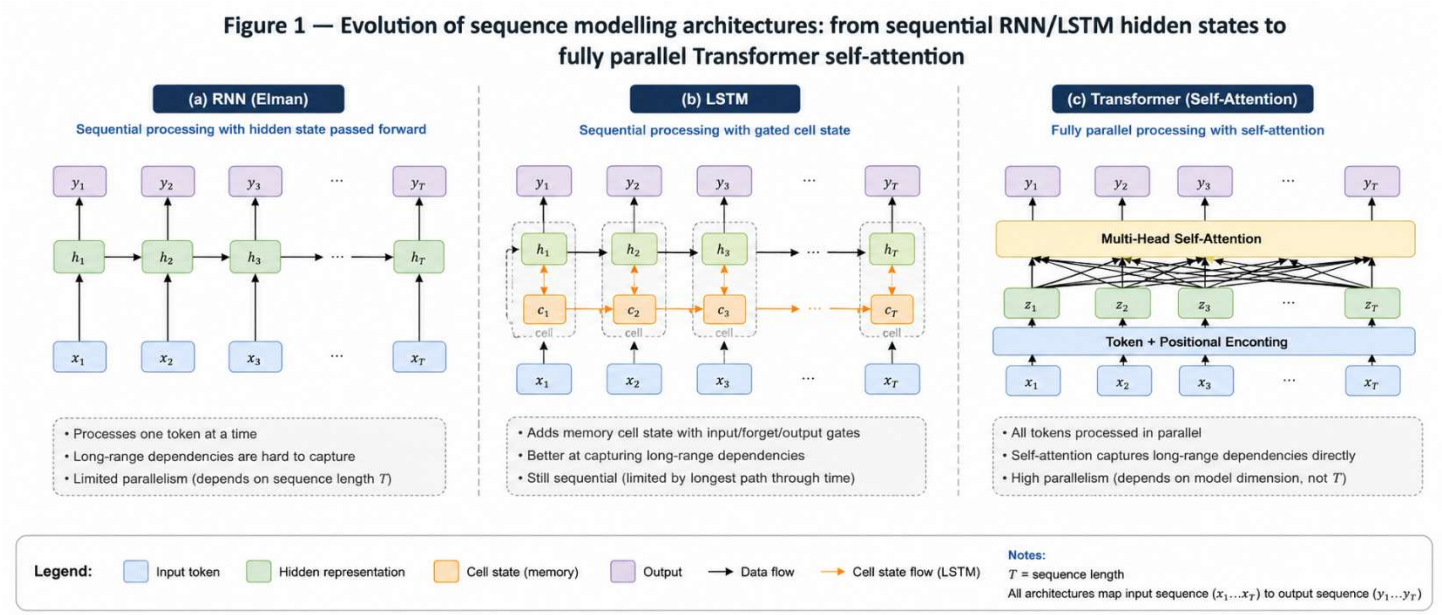



**Figure 1** — *Evolution of sequence modelling architectures: from sequential RNN/LSTM hidden states to fully parallel Transformer self-attention*

This convergence is not incidental. Self-attention is parallelisable by construction, scales predictably with increased model capacity and training compute, and preserves the full context without lossy compression. These properties made it the natural architectural foundation as the field scaled language models from millions to hundreds of billions of parameters. However, scaling also surfaced a new systems-level constraint—one that does not manifest during training but becomes the dominant bottleneck during inference.

## 1.3 A New Bottleneck: Memory Bandwidth During Inference

Solving the training-time limitations of recurrent models did not resolve all systems-level challenges; it deferred them to inference. The same property that makes self-attention effective during training—its ability to attend to the full sequence within each layer—creates a distinct and increasingly costly problem during autoregressive generation.

Decoder-only language models generate text one token at a time. At each step, the model processes the most recently produced token and outputs a probability distribution over the next. To compute attention over the full context at each decoding step, the model requires access to intermediate representations of every previously generated token. Recomputing these representations from scratch at each step would be computationally prohibitive for long sequences. The standard engineering solution is to cache these representations across decoding steps—specifically, the Key and Value tensors produced for each token in every attention layer. This structure is referred to as the Key-Value (KV) cache.

The KV cache grows monotonically throughout a generation pass. Each new token appended to the sequence contributes one Key tensor and one Value tensor per attention head per layer. For a model with many layers and many attention heads operating over a long sequence or a large batch of concurrent requests, this cache can occupy a substantial fraction of available GPU high-bandwidth memory (HBM). Under production serving conditions, where multiple sequences are decoded simultaneously, the

aggregate memory footprint of the KV cache scales accordingly and can exceed the memory available for model parameters.

The critical systems-level observation concerns arithmetic intensity. During training or prompt prefilling, the model processes large batches of tokens simultaneously, performing matrix multiplications of sufficient size to keep arithmetic units well-utilised. During autoregressive decoding, each forward pass processes only a single new token while reading the full KV cache for every layer. The ratio of floating-point operations to bytes of memory transferred is far lower. As the KV cache grows with sequence length and batch size, inference transitions from being compute-bound—limited by arithmetic throughput—to being memory-bandwidth-bound, limited by the rate at which data can be transferred between HBM and compute units. In this regime, reducing floating-point operation counts yields diminishing returns; reducing the volume of data that must be read from HBM at each step directly addresses the binding constraint.

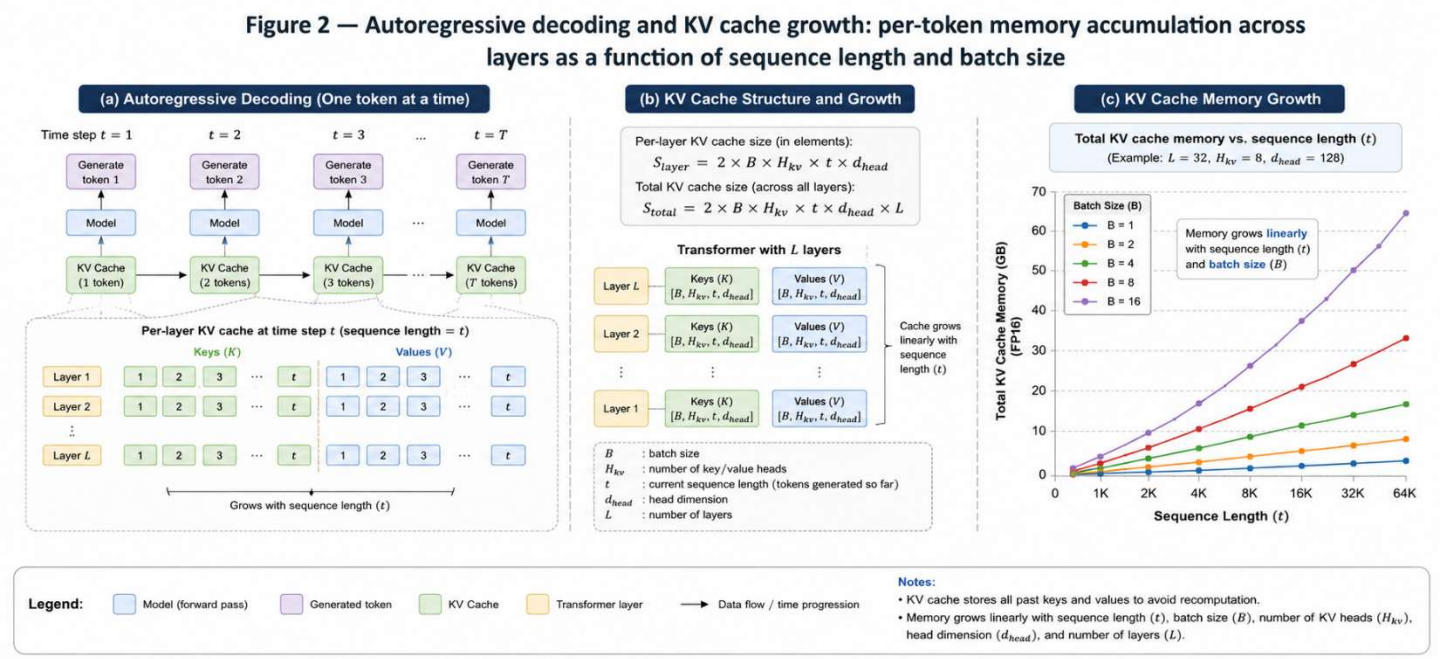



**Figure 2** — *Autoregressive decoding and KV cache growth: per-token memory accumulation across layers as a function of sequence length and batch size*

This shift from compute-bound to memory-bandwidth-bound behaviour as a function of sequence length is the central engineering motivation for the evolution of attention mechanisms examined in this report. Each successive mechanism represents a deliberate attempt to reduce the memory footprint of the KV cache, with distinct implications for inference throughput and model quality.

**ENGINEERING MOTIVATION**

My first encounter with this bottleneck was not theoretical—it arose from a production failure. While deploying an open-source language model behind a REST inference endpoint on a single NVIDIA T4 GPU using Hugging Face Transformers, the server behaved as expected under light load: individual requests completed within acceptable latency bounds, and memory utilisation appeared stable.

Under concurrent load, however, the inference server consistently raised CUDA out-of-memory errors. The initial diagnosis was straightforward: the model weights were too large for the available VRAM. After profiling memory allocation during live decoding, it became clear that model parameters were not the primary contributor to memory exhaustion. The KV cache—accumulating Key and Value tensor entries for each generated token across every layer of the model—was consuming GPU memory faster than requests could be serviced. Reducing batch size provided temporary relief, but did not address the underlying constraint.

That experience prompted a systematic study of how attention mechanisms have been redesigned to reduce KV cache memory consumption while preserving model quality. The present report is the result of that investigation.

Modern attention mechanisms should not be viewed as isolated architectural innovations. They represent successive engineering responses to a common systems-level constraint: the cost of maintaining and repeatedly reading a growing KV cache during autoregressive inference. Each mechanism in the progression—from Multi-Head Attention through Multi-Query Attention, Grouped Query Attention, and Multi-Head Latent Attention—addresses this constraint differently, with distinct trade-offs in memory consumption, computational overhead, and output quality. This observation motivates the remainder of the report, which examines each mechanism in turn.

# 2. Multi-Head Attention

The preceding chapter established that as language models scale and sequence lengths grow, autoregressive inference gradually transitions from a compute-bound regime to one dominated by memory bandwidth. The mechanism responsible for this transition is the Key-Value cache: a data structure that grows linearly with context length and must be read from GPU high-bandwidth memory at every decoding step. This chapter examines the attention mechanism that produces this structure in its original form—Multi-Head Attention (MHA) as specified by Vaswani et al. (2017)—and derives from first principles why it generates the memory access patterns that constrain inference throughput at scale.

## 2.1 The Multi-Head Attention Mechanism

The attention mechanism introduced by Vaswani et al. (2017) was motivated by a representational concern. A single attention computation applied over the full model dimension constrains the model to learn one global weighting of token relationships per layer. Yet the relationships relevant to language understanding are structurally diverse: a token may participate simultaneously in local syntactic dependencies, long-range semantic co-references, and positional patterns. Representing all of these within a single attention computation requires a learned compromise weighting that may capture none of them with high precision.

Multi-Head Attention resolves this by partitioning the total model dimension into h independent subspaces, each associated with a separate attention computation. Each subspace is termed an attention head. Every head maintains its own learned projection matrices for Queries, Keys, and Values and computes attention independently over the full input sequence. The resulting per-head outputs are then concatenated and projected back to the model dimension through a shared output matrix.

The representational advantage of this construction is that different heads can specialise during training. Analysis of trained Transformer models has confirmed that individual attention heads exhibit consistent specialisation patterns: some reliably focus on local syntactic structure, others track long-range co-references, and others encode relative positional relationships (Clark et al., 2019; Voita et al., 2019). This specialisation emerges from the training objective rather than from explicit constraints. Multi-Head Attention therefore provides richer representational capacity than a single-head mechanism of equivalent total dimension: the same parameter budget is distributed across independent subspaces, each free to specialise on a different aspect of the sequence.

Critically for inference, each head maintains its own Key and Value projection matrices, independent of every other head. This is the architectural choice that determines the size and growth rate of the KV cache.

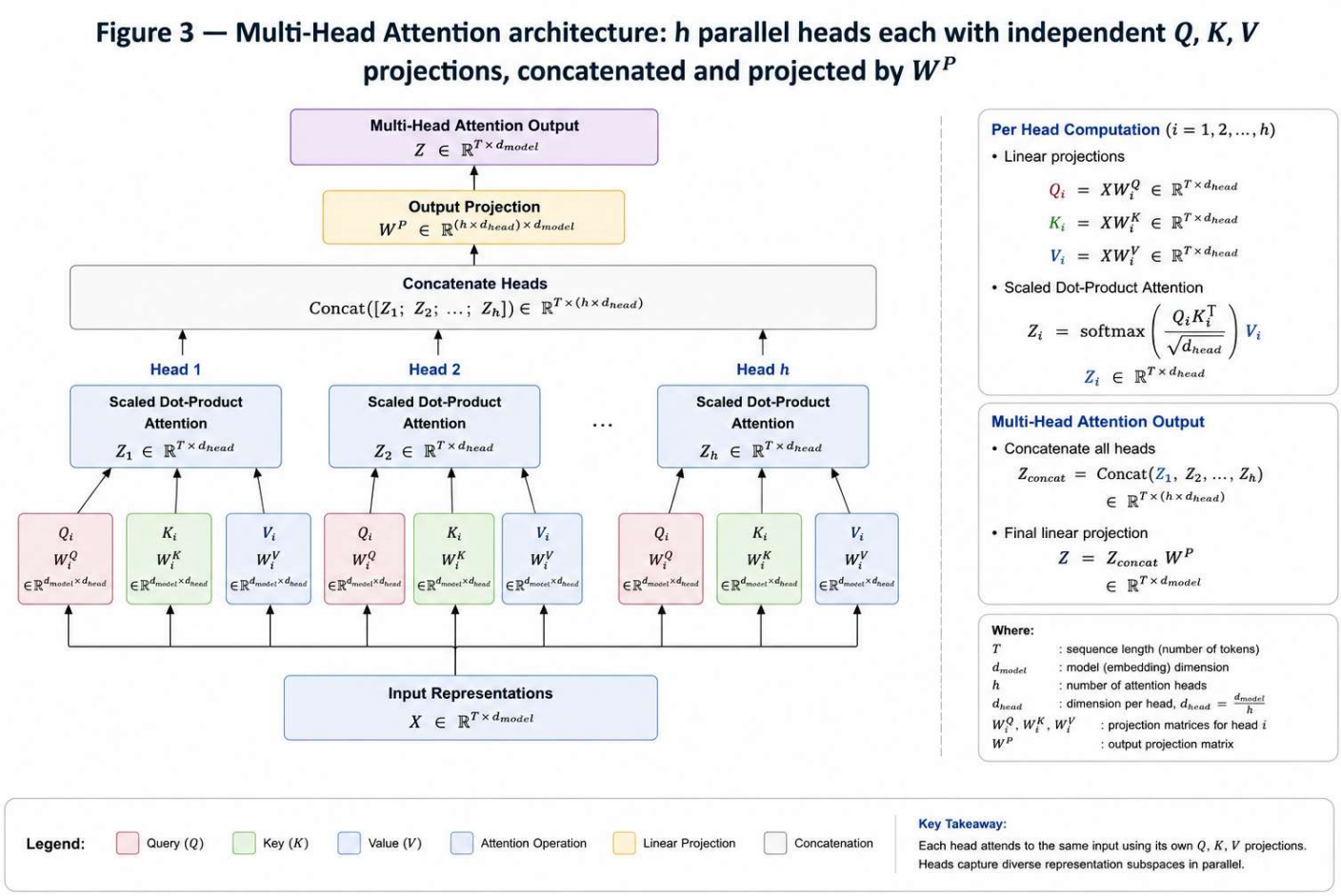



**Figure 3** — *Multi-Head Attention architecture: h parallel heads each with independent Q, K, V projections, concatenated and projected by $W^P$*

## 2.2 Mathematical Formulation

Let the input to an attention layer be a sequence of n tokens, each represented as a $d_{model}$-dimensional vector, collected into a matrix $X \in \mathbb{R}^{n \times d_{model}}$. For each attention head $i \in \{1, \ldots, h\}$, three linear projections transform X into the Query, Key, and Value matrices for that head:

$$Q_i = X W_i^Q \quad (1)$$

$$K_i = X W_i^K \quad (2)$$

$$V_i = X W_i^V \quad (3)$$

where $W_i^Q$, $W_i^K \in \mathbb{R}d_{model}xd_k$ and $W_i^V \in \mathbb{R}d_{model}xd_v$ are weight matrices unique to head $i$, learned during training. The projected dimension is typically set to $d_k = d_{model} / h$, so that the parameter count across all heads remains comparable to a single full-dimensional projection. The result is that each head operates in a lower-dimensional subspace, attending to relationships that are salient in that projection.

Attention within each head is computed by Scaled Dot-Product Attention:

$$\text{Attention}(Q, K, V) = \text{softmax} \quad (Q_i K_i^T / \sqrt{d_k}) \cdot V_i \tag{4}$$

The matrix product $Q_i K_i^T$ measures the pairwise compatibility between every Query vector and every Key vector in the sequence, yielding an ($n \times n$) score matrix. Scaling by $1/\sqrt{d_k}$ prevents the dot products from growing large in magnitude as $d_k$ increases, which would otherwise concentrate the softmax distribution near extreme values and suppress gradient flow during training (Vaswani et al., 2017). The softmax operation normalises the scaled scores into attention weights, which form a weighted sum over the Value vectors.

Each head produces an output matrix $\text{head}_i \in \mathbb{R}^{n \times d_v}$. The outputs of all $h$ heads are concatenated along the feature dimension and projected through a learned output matrix:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\, W^O \tag{5}$$

where $W^O \in \mathbb{R}^{h \cdot d_v \times d_{model}}$ maps the concatenated head outputs back to the model dimension, producing the attention output that is passed to the subsequent feedforward sub-layer.

**Table 1** — *Notation reference for equations (1)–(7)*

| Symbol | Dimension | Description |
|---|---|---|
| $X$ | $n \times d_{\text{model}}$ | Input token representation matrix |
| $Q_i$, $K_i$, $V_i$ | $n \times d_k$ or $d_v$ | Query, Key, and Value matrices for attention head i |
| $W_i^Q$, $W_i^K$ | $d_{\text{model}} \times d_k$ | Learned projection matrices for Queries and Keys of head i |
| $W_i^V$ | $d_{\text{model}} \times d_v$ | Learned projection matrix for Values of head i |
| $W^O$ | $hd_v \times d_{\text{model}}$ | Output projection matrix |
| $d_{\text{model}}$ | scalar | Model (embedding) dimension |
| $d_k$, $d_v$ | scalar | Projected dimension per head (typically d_model / h) |
| $h$ | scalar | Number of attention heads |
| $n$ | scalar | Input sequence length (number of tokens) |
| $L$ | scalar | Number of Transformer layers |
| $T$ | scalar | Context length (tokens generated so far) |
| $d_h$ | scalar | Head dimension: d_model / h |
| $p$ | scalar | Bytes per stored element (2 for FP16, 4 for FP32) |

## 2.3 Autoregressive Decoding: A Worked Example

Equations (1) through (5) describe the full attention computation during training, where all n tokens in the input are processed simultaneously in a single forward pass. During autoregressive inference, the computation proceeds fundamentally differently: tokens are generated one at a time, each conditioned on all tokens generated so far. Understanding the exact sequence of operations at each decoding step is essential to understanding why the KV cache exists and how it grows.

Consider generating the sequence "Hi how are you" from the initial input token "Hi." The following steps trace the operations performed at each decoding step across all L Transformer layers and all h attention heads.

### *Step 1: Initial Token — "Hi"*

The token "Hi" is mapped to its embedding vector and passed through all L Transformer layers in a forward pass. In each layer's attention module, the linear projections of equations (1) through (3) are applied to produce $Q_1$, $K_1$, and $V_1$ for each of the h attention heads. Since only a single token is present, the attention score reduces to a trivially weighted sum of $V_1$: the token attends to itself.

Following this forward pass, $K_1$ and $V_1$ for every head in every layer are written to the KV cache. These cached tensors encode the model's Key and Value projections of "Hi" at this stage of the network. The forward pass produces a distribution over the vocabulary; the token "how" is selected as the next token.

The Query $Q_1$ is not cached. The Query encodes the attending perspective of the current decoding step—what the model is searching for in the context—not the contextual information itself. Future tokens will compute their own Query vectors from their own embeddings and will have no use for the Query of "Hi." Only the Keys and Values, which encode what information is available to be attended to, are retained for future steps.

### *Step 2: Generating "how"*

The model receives the embedding of "how" as its current input. The critical engineering observation is that only the Query vector for the current token must be computed from scratch. The Keys and Values of all previous tokens—in this case, $K_1$ and $V_1$ for "Hi"—are read directly from the KV cache rather than being recomputed.

For each head and each layer, the model computes $Q_2$, $K_2$, and $V_2$ for the current token. $K_2$ and $V_2$ are immediately appended to the KV cache. Attention is then computed between $Q_2$ and the complete set of cached Keys:

$$A_t = \text{softmax}(Q_t K_{1:t}^T / \sqrt{d_k}) \cdot V_{1:t} \tag{6}$$

The output of this attention computation produces the next token, "are." The KV cache now holds four tensor entries per head per layer: $K_1$, $V_1$, $K_2$, $V_2$.

### *Step 3: Generating "are"*

The pattern repeats. The model computes $Q_3$, $K_3$, and $V_3$ for the token "are." $K_3$ and $V_3$ are appended to the cache, which now holds six entries per head per layer. Attention is computed between $Q_3$ and the three cached Key tensors [$K_1$; $K_2$; $K_3$], and the output selects the next token, "you."

The KV cache state at each step of this example is as follows. After "Hi": 2 entries per head per layer. After "how": 4 entries. After "are": 6 entries. After "you": 8 entries. Each generated token contributes exactly one Key tensor and one Value tensor per head per layer to the cache. For a sequence of T tokens across h heads and L layers, the total number of cached tensors is 2 × L × h × T.

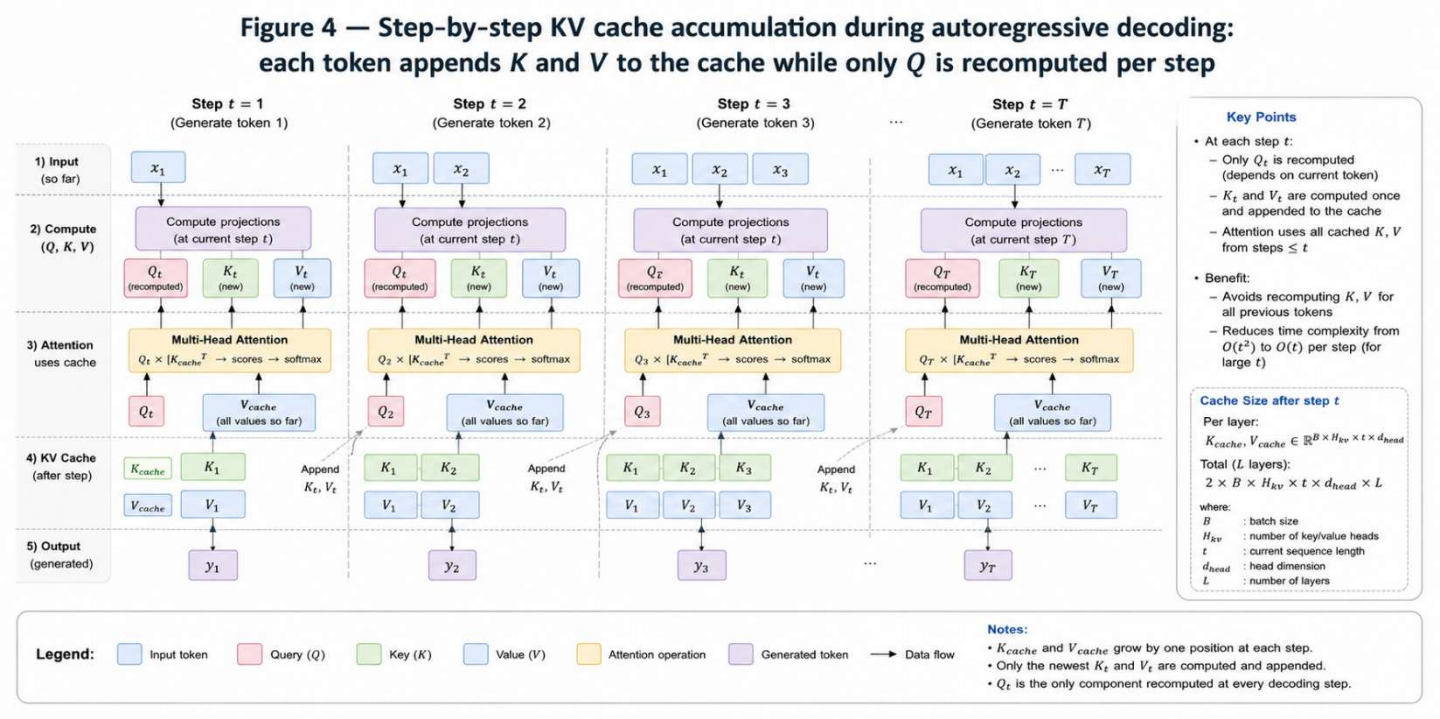



**Figure 4** — *Step-by-step KV cache accumulation during autoregressive decoding: each token appends K and V to the cache while only Q is recomputed per step*

### *Step 4: The Cost at Scale*

At a sequence length of 10,000 tokens, each decoding step requires reading 10,000 Key vectors and 10,000 Value vectors for every attention head in every Transformer layer from GPU HBM before any arithmetic computation can proceed. The arithmetic performed on this data—dot products between a single Query vector and 10,000 Key vectors for each head—is minimal relative to the volume of data transferred. The bottleneck is no longer arithmetic throughput; it is memory bandwidth. This is the hardware consequence of the KV cache design.

## 2.4 Growth of the KV Cache

The memory footprint of the KV cache at sequence length T can be derived precisely from the model's architectural parameters. Two tensors—one for Keys, one for Values—are stored for every head in every layer at every sequence position:

$$\text{KV Cache Size} = 2 \times L \times H_{kv} \times T \times d_h \times p \tag{7}$$

The factor of 2 accounts for the separate Key and Value storage matrices. L is the number of Transformer layers; $H_{kv}$ is the number of KV heads, which in Multi-Head Attention equals the total number of attention heads H; T is the current context length in tokens; $d_h$ is the head dimension, equal to $d_{model}$ / H; and p is the number of bytes required to store each element, determined by numerical precision. Every factor except T is a fixed architectural constant; T grows monotonically throughout a generation pass.

Two properties of equation (7) are noteworthy from a systems perspective. First, the cache scales linearly with context length: a 10× increase in context length produces a 10× increase in KV cache memory. Second, in Multi-Head Attention specifically, the cache also scales linearly with the number of heads $H_{kv}$ = H. As will be seen in subsequent chapters, it is precisely this second dependence that MQA and GQA reduce, and that MLA eliminates entirely through a different mechanism.

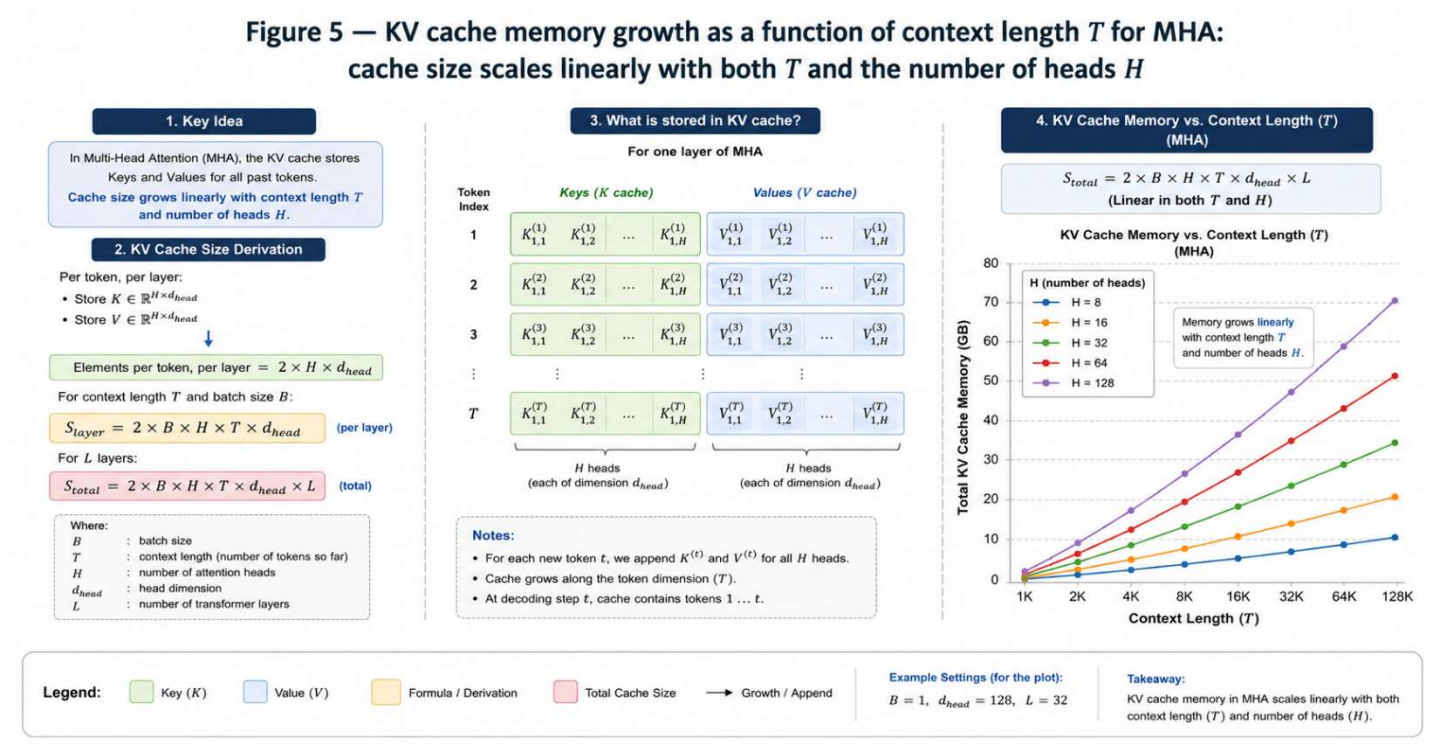



**Figure 5** — *KV cache memory growth as a function of context length T for MHA: cache size scales linearly with both T and the number of heads H*

## 2.5 Numerical Example

The following example applies equation (7) to the architecture of LLaMA-2-7B (Touvron et al., 2023), a widely referenced open-weight decoder-only model with standard Multi-Head Attention, to derive the KV cache memory requirement at the model's full context window.

**Table 2** — *KV cache memory calculation for LLaMA-2-7B at full context length (FP16 precision)*

| Parameter | Symbol | Value | Notes |
|---|---|---|---|
| Transformer layers | $L$ | 32 | LLaMA-2-7B architecture |
| Attention heads (KV) | $H_{kv}$ | 32 | Equals total heads in MHA |
| Head dimension | $d_h$ | 128 | $d_{model} / h = 4096 / 32$ |
| Context length | $T$ | 8,192 tokens | Maximum supported context |
| Bytes per element | $p$ | 2 bytes | FP16 precision |
| KV cache size | — | 4.0 GiB | 2 × 32 × 32 × 8192 × 128 × 2 bytes |

Substituting the values from Table 2 into equation (7):

$$\text{KV Cache Size} = 2 \times 32 \times 32 \times 8{,}192 \times 128 \times 2 \text{ bytes} = 4{,}294{,}967{,}296 \text{ bytes} \approx 4.0 \text{ GiB} \quad ()$$

This result places the KV cache size in context. The 7 billion parameters of LLaMA-2-7B in FP16 occupy approximately 14 GiB. A single inference request at the full context window of 8,192 tokens therefore requires approximately 4 GiB for the KV cache alone—representing 29% of the model's own weight memory. Under concurrent serving conditions with multiple simultaneous requests, the aggregate KV cache footprint scales proportionally with batch size. On a GPU with 16 GiB of HBM—a common configuration at deployment—model weights and a single full-context inference request already consume approximately 18 GiB of a 16 GiB budget, necessitating either context truncation, reduced batch sizes, or architectural modifications to the attention mechanism.

## 2.6 Hardware Perspective: The GPU Memory Hierarchy

To understand why the growing KV cache produces a hardware bottleneck, it is necessary to trace the path that attention data travels during each decoding step through the GPU memory hierarchy.

Modern inference GPUs—such as the NVIDIA A100 and H100—organise memory into two distinct tiers with fundamentally different capacity and bandwidth characteristics. The first tier is High Bandwidth Memory (HBM), which serves as the primary off-chip storage for all persistent data: model weights, intermediate activations, input and output buffers, and the KV cache. The A100 SXM4 provides 80 GiB of HBM2e at a peak bandwidth of approximately 2,000 GB/s; the H100 SXM5 provides 80 GiB of HBM3 at

approximately 3,350 GB/s. HBM bandwidth, while high relative to CPU memory, is a finite resource that multiple concurrent operations must share.

The second tier is on-chip memory, distributed across the GPU's Streaming Multiprocessors (SMs). Each SM contains registers, shared memory (SRAM), and an L1 cache; these on-chip resources support extremely high bandwidth access but are orders of magnitude smaller in capacity than HBM. On the A100, each SM provides 192 KB of configurable shared memory; across all 108 SMs, the total on-chip SRAM capacity is approximately 20 MB. Tensor Cores within each SM perform the matrix multiply-accumulate operations that constitute the bulk of transformer arithmetic.

The KV cache is too large to reside in on-chip SRAM. It lives in HBM. During every decoding step, the attention computation for the new token requires the following sequence of data movements: the Query vector for the current token is loaded from HBM into on-chip registers; all Key vectors for the accumulated context are streamed from HBM through the SMs; the compatibility scores are computed on the Tensor Cores; all Value vectors are streamed from HBM; and the weighted sum is computed to produce the attention output. Steps involving Key and Value streaming must transfer data proportional to T from HBM at every step. This repeated streaming—not the arithmetic computation itself—is the operation that becomes the binding constraint.

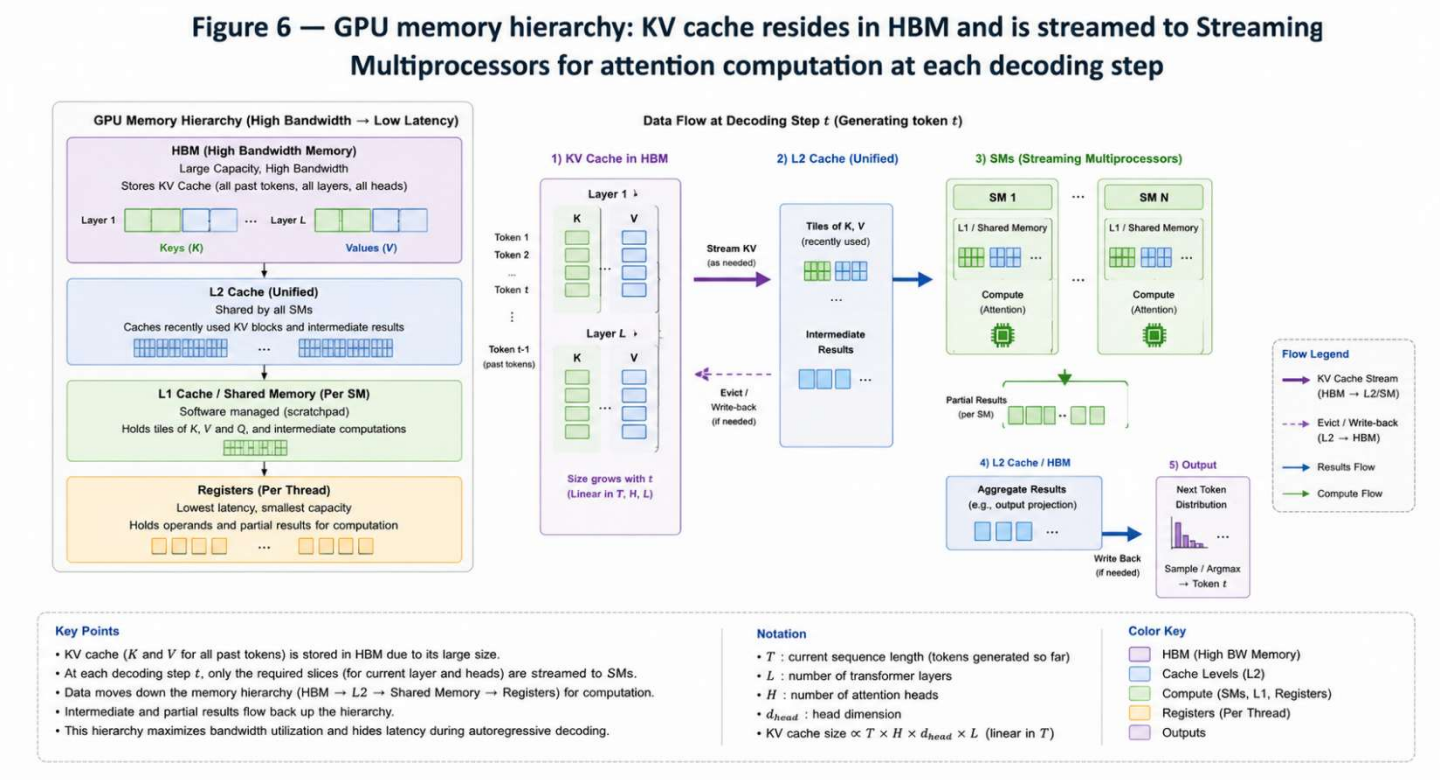



**Figure 6** — *GPU memory hierarchy: KV cache resides in HBM and is streamed to Streaming Multiprocessors for attention computation at each decoding step*

## 2.7 The Memory Bandwidth Bottleneck

The hardware consequences of the KV cache can be characterised quantitatively using the concept of arithmetic intensity: the ratio of floating-point operations to bytes of memory transferred. A hardware unit operates efficiently when arithmetic intensity is high enough to saturate arithmetic throughput. When arithmetic intensity is low, arithmetic units idle while waiting for data.

During training and prompt prefilling, the model processes large batches of tokens simultaneously. Matrix multiplications operate on large tensors, arithmetic intensity is high, and Tensor Core utilisation approaches its theoretical peak. During autoregressive decoding, each forward pass processes only a single new token while reading the full KV cache. For a model with 32 heads and head dimension 128 decoding at a context length of 8,192 tokens, the attention computation for a single layer requires reading approximately 128 MB of Key and Value data from HBM (32 heads × 8,192 tokens × 128 dimensions × 2 tensors × 2 bytes), while performing on the order of tens of millions of floating-point operations. The resulting arithmetic intensity is well under 1 FLOP per byte of data transferred.

The NVIDIA A100 roofline threshold—the arithmetic intensity above which an operation is compute-bound rather than bandwidth-bound—is approximately 156 FLOPs per byte (312 TFLOPS FP16 / 2,000 GB/s bandwidth). Autoregressive decoding at long context lengths operates two to three orders of magnitude below this threshold, confirming that such workloads are deeply memory-bandwidth-bound. Increasing GPU compute throughput provides negligible improvement; only reducing the volume of data transferred per step—equivalently, reducing the KV cache size—materially improves inference latency.

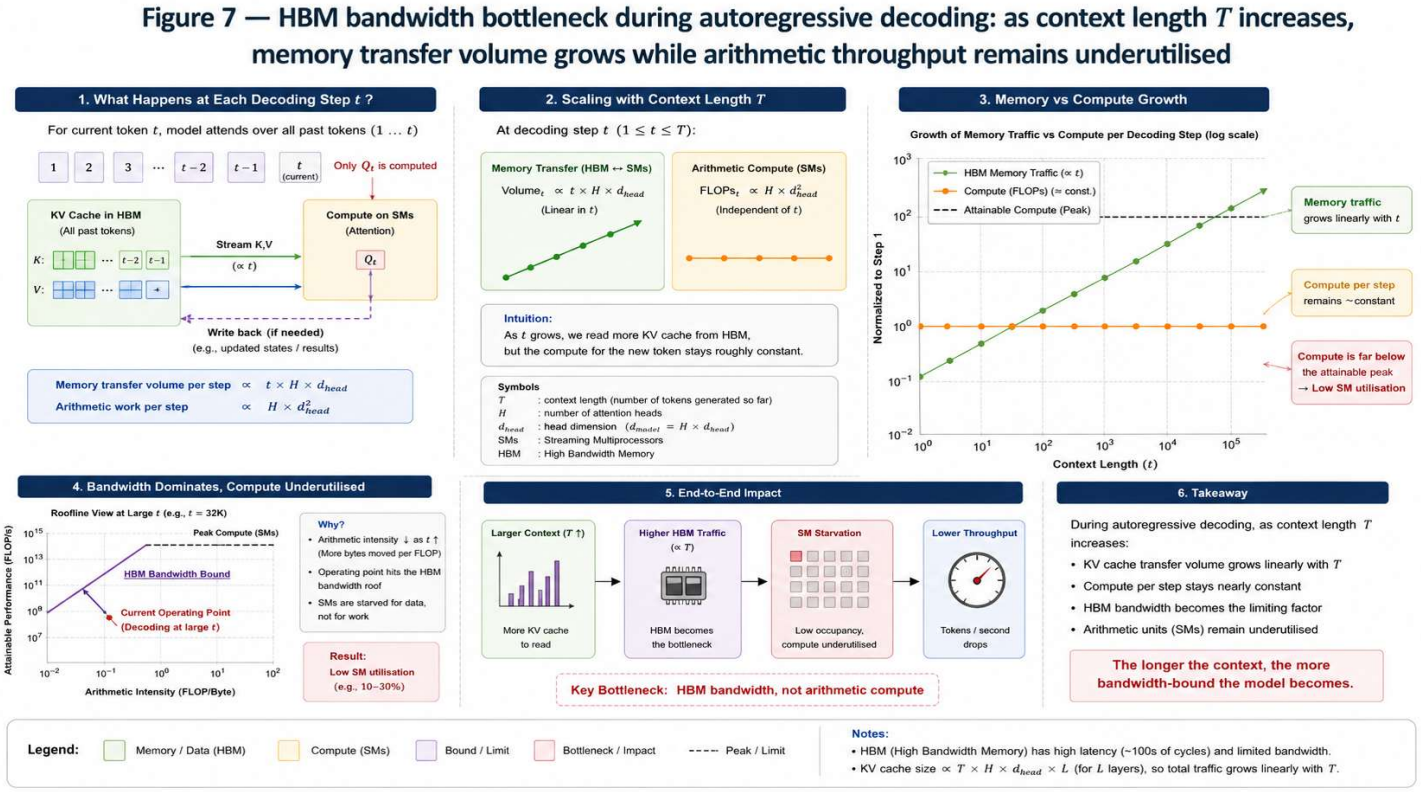



**Figure 7** — *HBM bandwidth bottleneck during autoregressive decoding: as context length T increases, memory transfer volume grows while arithmetic throughput remains underutilised*

The practical consequence is a latency curve that increases approximately linearly with context length during decoding. For a model serving interactive requests, this means that generating the 5,000th token takes materially longer than generating the 500th, not because additional computation has been introduced, but because more data must be read from HBM at each step.

## 2.8 Engineering Limitations of Multi-Head Attention

Multi-Head Attention provides the highest representational capacity of any attention variant examined in this report. Every head maintains fully independent Query, Key, and Value projection matrices, enabling the model to learn h orthogonal representations of the input sequence simultaneously. Empirically, this design produces attention heads with distinct specialisation patterns, suggesting that the independent projections contribute meaningfully to model quality. MHA is theoretically well-motivated and empirically effective.

However, MHA also produces the largest possible KV cache. Because every head stores independent Key and Value representations for every token in the context, and because these representations must be read from HBM at every decoding step, the per-step memory transfer cost scales as $2 \times L \times H \times d_h \times T$ bytes. Both L and H are proportional to model size; T grows throughout every inference request. At scales typical of production language models—dozens of layers, many attention heads, and context windows extending to tens of thousands of tokens—the KV cache dominates inference memory consumption and caps achievable throughput.

This analysis motivates a precise engineering question: does each of the H attention heads require its own independent Key and Value projections to maintain acceptable output quality? The Query projection must remain unique per head, since the Query encodes the attending perspective and different heads must attend to different patterns. But the Keys and Values encode the information being attended to—the content of the context. If multiple Query heads could share a single set of Key and Value projections without significant degradation in model quality, the KV cache size would be reduced by a factor of H, with a corresponding reduction in the per-step HBM bandwidth requirement.

This question was directly and affirmatively answered by Shazeer (2019) in the proposal of Multi-Query Attention. Rather than maintaining independent Key and Value projections per head, Multi-Query Attention uses a single shared set of Keys and Values across all Query heads, reducing the KV cache to 1/H of its Multi-Head Attention size at the cost of some representational capacity. The nature of that trade-off, and its engineering implications, are examined in the following chapter.

# 3. Multi-Query Attention

The preceding chapter established that Multi-Head Attention imposes an HBM bandwidth cost that scales as $2 \times L \times H \times T \times d_h \times p$ bytes per decoding step, where H is the number of attention heads and T grows monotonically throughout every inference request. At the scales typical of production language models—thirty-two layers, thirty-two heads, and context windows extending to tens of thousands of tokens—this cost dominates inference memory consumption and constrains achievable throughput. Multi-Query Attention (MQA), introduced by Shazeer (2019), addresses this bottleneck by challenging an implicit assumption embedded in the original Multi-Head Attention design: that independent Key and Value projections are necessary for each attention head.

## 3.1 The KV Sharing Hypothesis

In Multi-Head Attention, each of the H attention heads maintains its own learned Key projection matrix and Value projection matrix, producing per-head Key and Value tensors independently for every input token. The resulting KV cache must store H distinct Key tensors and H distinct Value tensors per token per layer, and every one of these tensors must be read from HBM at each decoding step.

Shazeer (2019) proposed a fundamentally different design: while preserving H independent Query projections—one per head—collapse all Key and Value projections to a single shared set. Every Query head, regardless of its specialisation, attends to the same Key tensor and retrieves from the same Value tensor. The KV cache therefore stores only one Key and one Value tensor per token per layer, regardless of the number of Query heads. This is the central hypothesis of Multi-Query Attention: that the Query projection alone carries the representational diversity necessary for effective multi-head attention, and that the Keys and Values can be shared without critical loss of model capability.

This hypothesis, if true, would reduce the KV cache by a factor of H, with a proportional reduction in HBM reads per decoding step. For a model with thirty-two attention heads, this represents a thirty-two-fold reduction in KV cache memory and a corresponding reduction in the primary bottleneck identified in Chapter 2.

## 3.2 Multi-Query Attention Architecture

The Multi-Query Attention mechanism is structurally identical to Multi-Head Attention with one critical modification: the number of Key and Value heads, Hkv, is fixed to one. The H Query projection matrices remain independent: each head i = 1, ..., H maintains its own learned weight matrix and projects the input sequence X into a head-specific Query matrix Qi.

In contrast, the Key and Value projections are each defined by a single matrix shared across all heads: W^K and W^V, respectively. These matrices produce a single Key tensor K and a single Value tensor V that are shared across all H Query heads during attention computation. The output of each head is then computed by attending the head-specific Query against the shared Key and Value, and the final output is formed by concatenating all head outputs and projecting through the shared output matrix W^O, exactly as in Multi-Head Attention.

During autoregressive decoding, the KV cache stores only K and V—one Key tensor and one Value tensor per token per layer—rather than H of each. The Query tensor Qi is not cached. As in Multi-Head Attention,

the Query is computed fresh for the current token at each decoding step, used immediately for the attention computation, and then discarded.

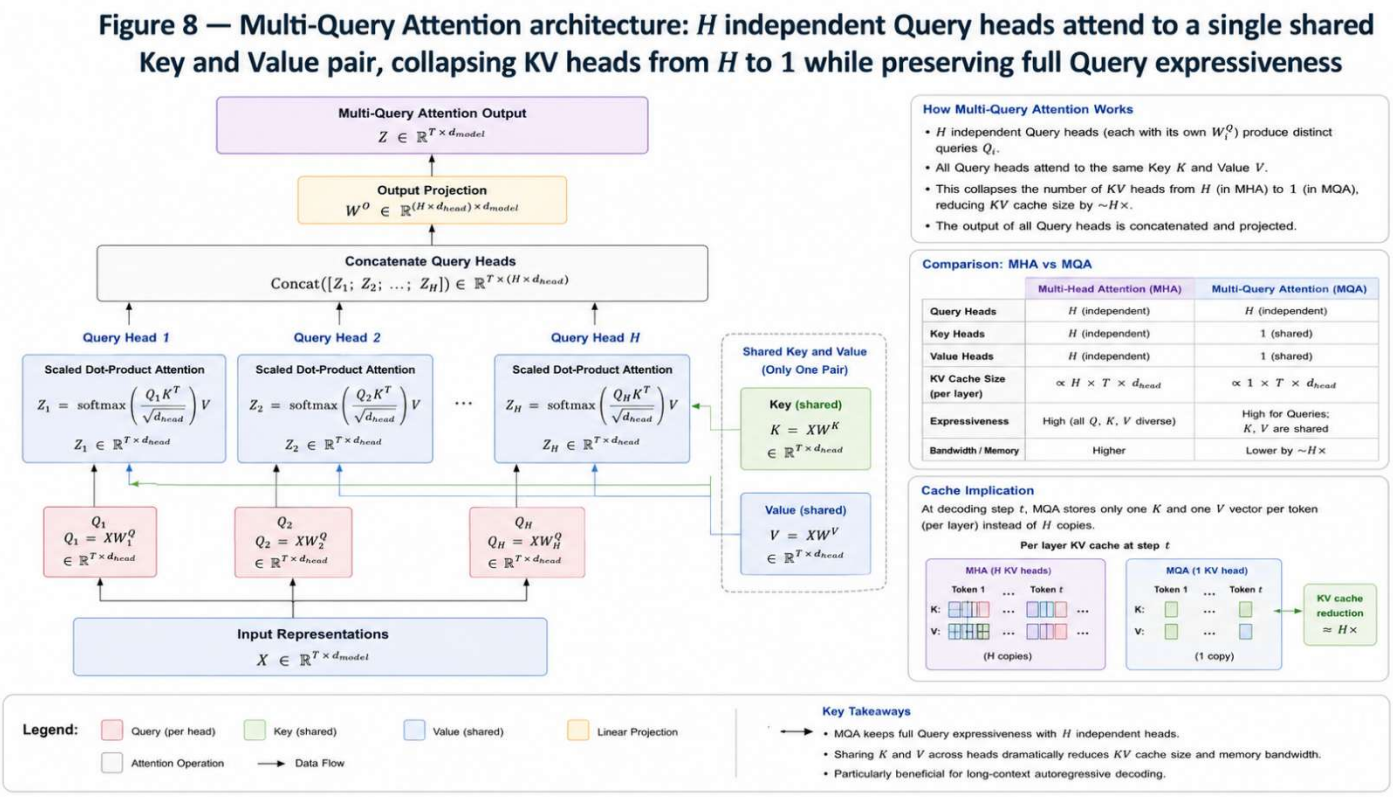



**Figure 8** — *Multi-Query Attention architecture: H independent Query heads attend to a single shared Key and Value pair, collapsing KV heads from H to 1 while preserving full Query expressiveness*

## 3.3 Mathematical Formulation

Let the input sequence be X of shape n × dmodel, where n is the sequence length and dmodel is the model dimension. For each head i = 1, ..., H, the Query projection is computed independently:

$$Q_i = X W_i^Q \tag{8}$$

The Key and Value projections are shared across all Query heads and are computed using a single pair of projection matrices:

$$K = X W^K \tag{9}$$

$$V = X W^V \tag{10}$$

Note the absence of a head index on K and V. Both tensors are produced once per layer from the full input X and are shared across all H Query heads. The attention computation for head i is then:

$$\text{Attention}_i = \text{softmax}(Q_i K^T / \sqrt{d_k}) \cdot V \tag{11}$$

Each Query head i produces its own attention weight distribution over the sequence by comparing Qi against the shared K. The resulting weighted sum draws from the shared V. The H head outputs are then concatenated and projected:

$$\text{MQA}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\, W^O \tag{12}$$

The output projection W^O is unchanged from Multi-Head Attention. The sole architectural difference is the elimination of the per-head subscript from K and V, and the replacement of H Key/Value projection matrices with one each.

**Table 3** — *Notation reference for Multi-Query Attention equations (8)–(12)*

| Symbol | Dimensions | Description |
|---|---|---|
| $Q_i$ | $n \times d_k$ | Query matrix for head i (independent per head) |
| $K$ | $n \times d_k$ | Shared Key matrix (no head index) |
| $V$ | $n \times d_v$ | Shared Value matrix (no head index) |
| $W_i^Q$ | $d_{model} \times d_k$ | Query projection for head i |
| $W^K$ | $d_{model} \times d_k$ | Shared Key projection |
| $W^V$ | $d_{model} \times d_v$ | Shared Value projection |
| $H$ | scalar | Total number of Query heads |
| $H_{kv}$ | 1 | Number of KV heads in MQA (fixed to 1) |

## 3.4 KV Cache Under Multi-Query Attention

Substituting Hkv = 1 into the KV cache formula derived in Chapter 2 (Equation 7) yields the KV cache footprint for Multi-Query Attention:

$$\text{KV Cache (MQA)} = 2 \times L \times 1 \times T \times d_h \times p \tag{13}$$

The factor H is replaced by 1. For a model with H = 32 attention heads such as LLaMA-2-7B, this represents a thirty-two-fold reduction in KV cache relative to Multi-Head Attention. Applying the numerical parameters from Table 2 of Chapter 2:

KV Cache (MQA) = 2 × 32 × 1 × 8,192 × 128 × 2 bytes = 134,217,728 bytes ≈ 128 MiB

In comparison, the same model under Multi-Head Attention requires approximately 4.0 GiB for the KV cache at full context. Multi-Query Attention reduces this to 128 MiB—a factor of 32. The corresponding reduction in HBM reads per decoding step is proportional, since the bandwidth cost at each step is determined by the volume of KV data streamed from HBM into the streaming multiprocessors.

**Table 4** — *KV cache comparison between MHA and MQA for LLaMA-2-7B at full context (T = 8,192, FP16)*

| Property | Multi-Head Attention | Multi-Query Attention |
|---|---|---|
| Query heads | 32 | 32 |
| KV heads ($H^{kv}$) | 32 | 1 |
| Keys stored per token per layer | 32 × 128 = 4,096 elements | 1 × 128 = 128 elements |
| Values stored per token per layer | 32 × 128 = 4,096 elements | 1 × 128 = 128 elements |
| KV cache at T = 8,192 (FP16) | ≈4.0 GiB | ≈128 MiB |
| KV cache reduction factor | 1× (baseline) | 32× smaller |
| HBM reads per decoding step | 4.0 GiB at T = 8,192 | 128 MiB at T = 8,192 |

**Figure 9 — KV cache growth comparison between MHA and MQA as a function of context length $T$: MQA grows at $1/H$ of the MHA rate, yielding a 32× reduction for a 32-head model at any context length**

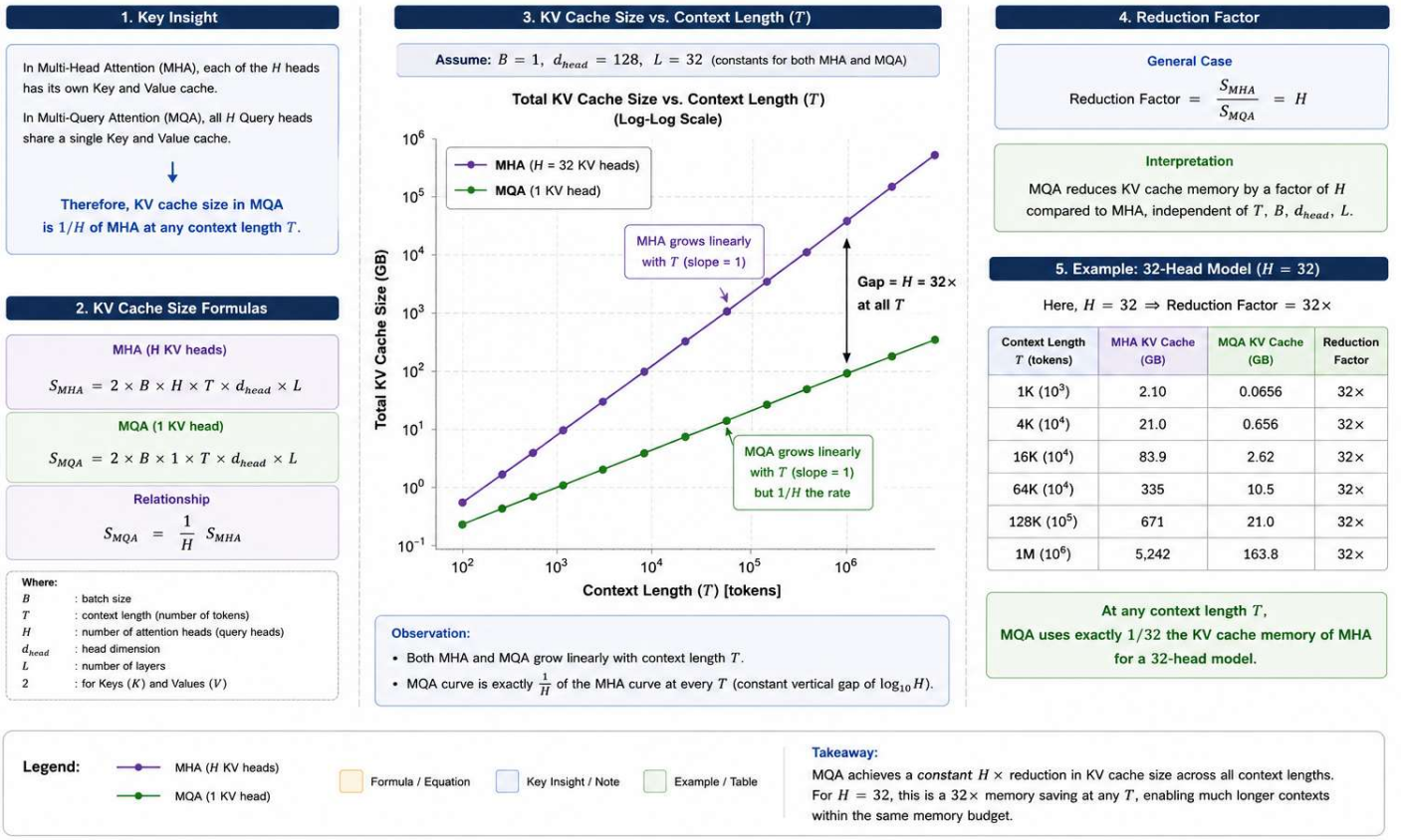



*Figure 9 — KV cache growth comparison between MHA and MQA as a function of context length T: MQA grows at 1/H of the MHA rate, yielding a 32× reduction for a 32-head model at any context length*

## 3.5 Autoregressive Decoding Under Multi-Query Attention

The autoregressive decoding process for Multi-Query Attention follows the same structure as Multi-Head Attention but with a substantially smaller KV cache. Continuing the worked example from Chapter 2, consider decoding the sequence 'Hi how are you' from the initial input 'Hi'.

***Step 1: Processing 'Hi'***

The embedding of 'Hi' passes through all transformer layers. In each layer, the model computes H = 12 independent Query vectors—one per head—using the H independent Query projection matrices. It also computes one Key vector $K_1$ and one Value vector $V_1$ using the single shared projection matrices. $K_1$ and $V_1$ are stored in the KV cache. The H Query vectors are used immediately for the attention computation and then discarded. The KV cache after Step 1 contains one Key-Value pair per layer.

***Step 2: Generating 'how'***

The embedding of 'how' is processed through all layers. Each layer computes H = 12 new Query vectors. It computes one new Key $K_2$ and one new Value $V_2$, appended to the KV cache. Each Query head then attends over the cached sequence [$K_1$, $K_2$] and retrieves weighted sums from [$V_1$, $V_2$]. Critically, all H = 12 Query heads attend to exactly the same two-token KV cache. Head 1 and Head 12 use different Q vectors to generate different attention patterns, but both attend to the same $K_1$, $K_2$ and retrieve from the same $V_1$, $V_2$.

***Steps 3–4: Generating 'are' and 'you'***

The pattern repeats. Each step appends one new Key-Value pair to the cache and requires all H Query heads to read the full cached history. After decoding 'you', the KV cache contains four Key-Value pairs per layer—compared to H × 4 = 48 pairs in Multi-Head Attention for H = 12. The HBM reads at each step scale with the number of cached KV pairs rather than the number of Query heads. Multi-Query Attention therefore decouples query-side representational capacity from KV cache memory: the same H Query heads operate at full capacity while the KV footprint is reduced by a factor of H.

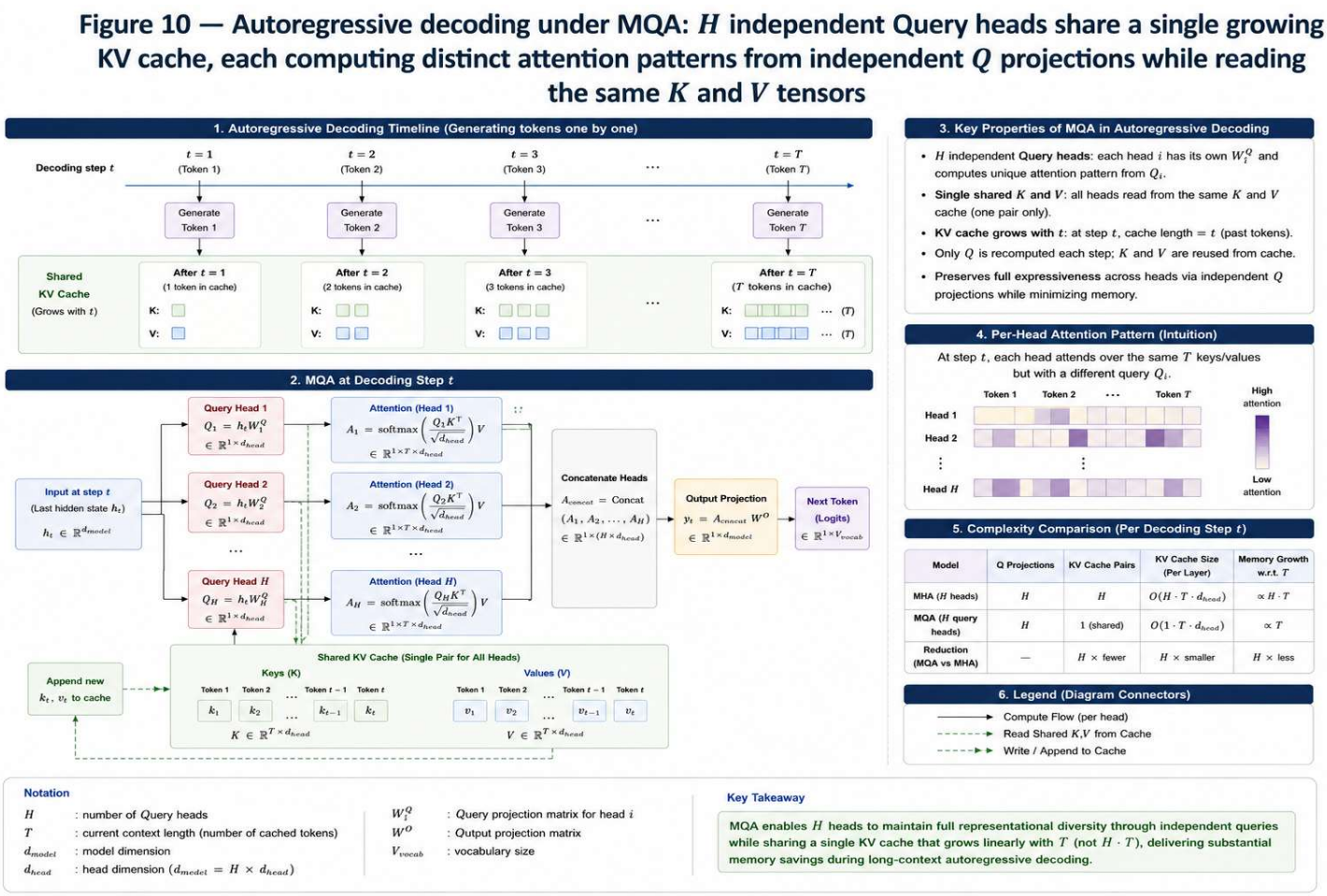



**Figure 10** — *Autoregressive decoding under MQA: H independent Query heads share a single growing KV cache, each computing distinct attention patterns from independent Q projections while reading the same K and V tensors*

## 3.6 Why Multi-Query Attention Reduces Model Quality

The KV cache reduction demonstrated in Section 3.4 comes at a cost. Empirical evaluations of models trained with Multi-Query Attention consistently show a measurable reduction in model quality relative to Multi-Head Attention baselines trained with otherwise identical hyperparameters. Understanding why this degradation occurs requires examining what the Key and Value projections actually encode.

In Multi-Head Attention, the Key projection $W^N K$ for head i is a learned transformation of the input sequence. Through training, it develops to encode the particular aspect of the sequence most relevant to the queries that head i specialises in. A head specialised in syntactic subject-verb dependencies will learn a Key projection that highlights the syntactic properties of tokens. A head specialised in long-range coreference will learn a Key projection that emphasises semantic entity identity across the sequence. These are structurally different transformations, and the optimal Key projection for one head is not the optimal projection for another.

The same argument applies to the Value projections. The Value $V^N$ determines which information is extracted from the sequence given the attention distribution. A syntactic head extracts different information from a token than a semantic head, even if both attend to the same positions—which they

generally do not. Per-head Key and Value projections are therefore not redundant: they are functional components that encode different aspects of the sequence for different head specialisations.

Multi-Query Attention eliminates per-head Key and Value projections. A single $W^PK$ must simultaneously serve as the Key projection for all H Query heads. This one transformation must encode the aspects of the sequence relevant to syntactic heads, semantic heads, positional heads, and any other specialisation that emerges during training—simultaneously. Since these requirements are structurally diverse and often conflicting, the single shared projection cannot satisfy all of them with high precision. It learns a compromise representation that captures some structure but not the depth of specialisation that H independent projections would achieve.

An analogy clarifies the problem. Consider twelve engineers working in different technical domains—compilers, networking, storage, graphics, cryptography, and so on. In Multi-Head Attention, each engineer has access to a domain-specific technical database curated precisely to their specialisation. The compiler engineer's database emphasises language theory and optimisation literature; the networking engineer's database emphasises protocol specifications and congestion-control research. In Multi-Query Attention, all twelve engineers are given a single shared database assembled as a compromise across all twelve domains. No single engineer receives a database curated to their precise needs. The quality of each engineer's work degrades because the information available to them is not optimised for their domain.

The Queries are not affected by this problem because they are not shared. Each head retains its own independent Query projection, meaning each head still asks different questions. The degradation arises because all twelve heads must look up their answers in the same shared Key and Value store, regardless of the specificity of their question.

In practice, the magnitude of the quality degradation depends on the model architecture, training duration, and task distribution. For some configurations, fine-tuning a pre-trained Multi-Head Attention model to use Multi-Query Attention via a brief additional training phase partially recovers the lost quality. Nevertheless, the fundamental representational limitation remains: a single shared Key and Value cannot provide the specialisation that per-head projections offer. This motivated the development of Grouped Query Attention.

## 3.7 Inference Efficiency Gains

Despite the quality trade-off, Multi-Query Attention delivers substantial inference efficiency improvements. The primary gain is a reduction in HBM bandwidth consumption per decoding step. Because the KV cache is H times smaller, each decoding step transfers H times less data from HBM into the streaming multiprocessors. For a model with H = 32 heads, this is a thirty-two-fold reduction in KV-related bandwidth demand.

The arithmetic intensity of the decoding operation—measured in floating-point operations per byte transferred—increases correspondingly. This shifts the operation further above the memory-bandwidth-bound regime identified in Chapter 2, translating directly to reduced per-token latency in memory-bandwidth-limited inference scenarios. A secondary benefit is increased serving throughput: because the KV cache is smaller, more concurrent inference requests can be served within the same GPU memory budget, improving aggregate system utilisation.

These gains are real and significant. However, the quality degradation relative to Multi-Head Attention has led the industry to adopt Grouped Query Attention as the preferred architecture rather than Multi-Query

Attention. Grouped Query Attention captures the majority of the efficiency gains while recovering most of the quality loss. The following chapter examines how this is achieved.

## 4. Grouped Query Attention

Multi-Query Attention demonstrated that sharing Key and Value projections across all Query heads yields dramatic KV cache reductions and corresponding improvements in inference throughput. The cost of this sharing is a measurable reduction in model quality, arising because a single Key-Value representation cannot serve the diverse specialisation demands of all Query heads simultaneously. Grouped Query Attention (GQA), introduced by Ainslie et al. (2023), addresses this limitation by introducing a middle ground: rather than one shared KV pair for all heads or one independent KV pair per head, a small number G of shared KV groups are maintained, each serving a distinct subset of Query heads.

### 4.1 Why Multi-Query Attention Was Not Enough

Multi-Query Attention established an important empirical finding: Key and Value projections contain considerable redundancy across attention heads, and this redundancy can be exploited by sharing to dramatically reduce KV cache size without destroying model utility. This was a meaningful contribution. However, collapsing all KV representations to a single shared pair proved too aggressive for general-purpose deployment.

The representational limitation is structural. In Multi-Query Attention, every Query head computes its attention weights by comparing its unique Qi vector against the same shared Key sequence K. The attention distribution differs across heads because the Qi vectors are head-specific. However, the Value sequence V from which each head retrieves information is identical. A head that has developed a syntactic specialisation retrieves from the same Value representation as a head that has specialised in long-range semantic associations. These heads would benefit from Value representations curated to their respective specialisations. A single shared V cannot provide this.

The consequence in trained models is a consistent quality gap relative to Multi-Head Attention. Across language modelling benchmarks and downstream evaluations, Multi-Query Attention models exhibit higher perplexity and lower task performance than Multi-Head Attention counterparts of equivalent parameter count. This gap narrows during extended fine-tuning but does not fully close, reflecting the structural constraint imposed by sharing a single Key and Value across all Query heads. Multi-Query Attention demonstrated the efficiency principle. Grouped Query Attention refines the mechanism to recover the lost quality.

### 4.2 A Natural Engineering Question

The design space between Multi-Head Attention and Multi-Query Attention can be parameterised by a single integer: the number of Key-Value heads, Hkv. Multi-Head Attention uses Hkv = H. Multi-Query Attention uses Hkv = 1. Both are extreme configurations. The question that directly motivates Grouped Query Attention is: what are the accuracy and efficiency characteristics at intermediate values of Hkv, where $1 < Hkv < H$?

Specifically: partition the H Query heads into G groups of equal size. Each group shares one Key and one Value projection, independently of every other group. The number of KV heads is G. If G = 1, the architecture reduces exactly to Multi-Query Attention. If G = H, it reduces exactly to Multi-Head Attention. By choosing G between these extremes, the designer trades KV cache size against representational diversity.

This question was answered by Ainslie et al. (2023), who demonstrated empirically that a small number of KV groups—typically G = H/4 or G = H/8—recovers most of the quality gap between Multi-Query Attention and Multi-Head Attention, while retaining most of the inference efficiency gains of Multi-Query Attention. This finding established Grouped Query Attention as the practical default for production language models.

## 4.3 Why Increasing the Head Dimension Was Not the Solution

Before examining the GQA mechanism itself, it is worth addressing a natural alternative that might arise when considering how to recover the quality lost by Multi-Query Attention. If the single shared Key and Value representation in MQA is insufficiently expressive, why not increase the dimensionality of that representation? If each shared Key and Value projects into a larger subspace, the representation becomes richer. Could a high-dimensional single Key and Value compensate for the loss of per-head specialisation?

The KV cache formula makes the answer precise. From Equation 13, the KV cache for Multi-Query Attention scales as $2 \times L \times 1 \times T \times d_h \times p$. If the head dimension $d_h$ is doubled—from 128 to 256 dimensions, for instance—the KV cache doubles. Rather than reducing the inference bottleneck, increasing the head dimension directly worsens it. The memory volume transferred from HBM per decoding step increases proportionally with $d_h$. The same holds for Multi-Head Attention: adding dimensions to any KV head increases the cache size linearly.

Furthermore, the expressiveness argument does not resolve the representational limitation. The problem with Multi-Query Attention is not that the shared representation is too low-dimensional. The problem is that one representation must simultaneously serve H Query heads whose optimal Key and Value projections are structurally diverse. Making that single shared representation higher-dimensional does not resolve the structural mismatch; it merely provides a larger space within which a single compromise must be found. The fundamental issue—that one projection must serve multiple incompatible specialisations—remains unchanged.

There is also a parameter count consideration. The Key and Value projection matrices scale with $d_h$ in both size and compute cost. Doubling $d_h$ doubles the parameters in these matrices, the compute required to form K and V during prefilling, and the size of every activation tensor in the attention path. Increasing dimensionality therefore increases model size and training cost alongside inference memory cost. This moves in precisely the opposite direction from the engineering objective.

The correct lever is not the dimension of each KV representation but the number of KV representations. Grouped Query Attention pulls this lever.

## 4.4 Grouped Query Attention Architecture

Grouped Query Attention partitions the H Query heads into G groups of equal size, where G divides H. Each group $g = 1, \ldots, G$ maintains its own shared Key projection matrix $W^GK$ and Value projection matrix $W^GV$. All H/G Query heads assigned to group g share the Key tensor $K^G$ and Value tensor $V^G$ produced by these projections. The H Query projection matrices remain fully independent: each head i retains its own $W^IQ$ regardless of group membership.

A critical distinction must be emphasised at the outset: the number of Query heads H is not reduced. Every Query head remains, and every Query head retains its own independent Query projection. Only the number of Key and Value projections changes, from H (as in MHA) to G (where $G < H$). Queries remain fully

expressive and head-specific. Only the Key and Value representations are shared, and they are shared only within groups rather than across all heads simultaneously.

To make the grouping concrete, consider H = 12 Query heads and G = 4 KV groups. The H/G = 3 Query heads in each group share one Key and one Value projection:

| Group | Query Heads | Shared Key | Shared Value |
|---|---|---|---|
| Group 1 | $Q_1$, $Q_2$, $Q_3$ | $K_1$ | $V_1$ |
| Group 2 | $Q_4$, $Q_5$, $Q_6$ | $K_2$ | $V_2$ |
| Group 3 | $Q_7$, $Q_8$, $Q_9$ | $K_3$ | $V_3$ |
| Group 4 | $Q_{10}$, $Q_{11}$, $Q_{12}$ | $K_4$ | $V_4$ |

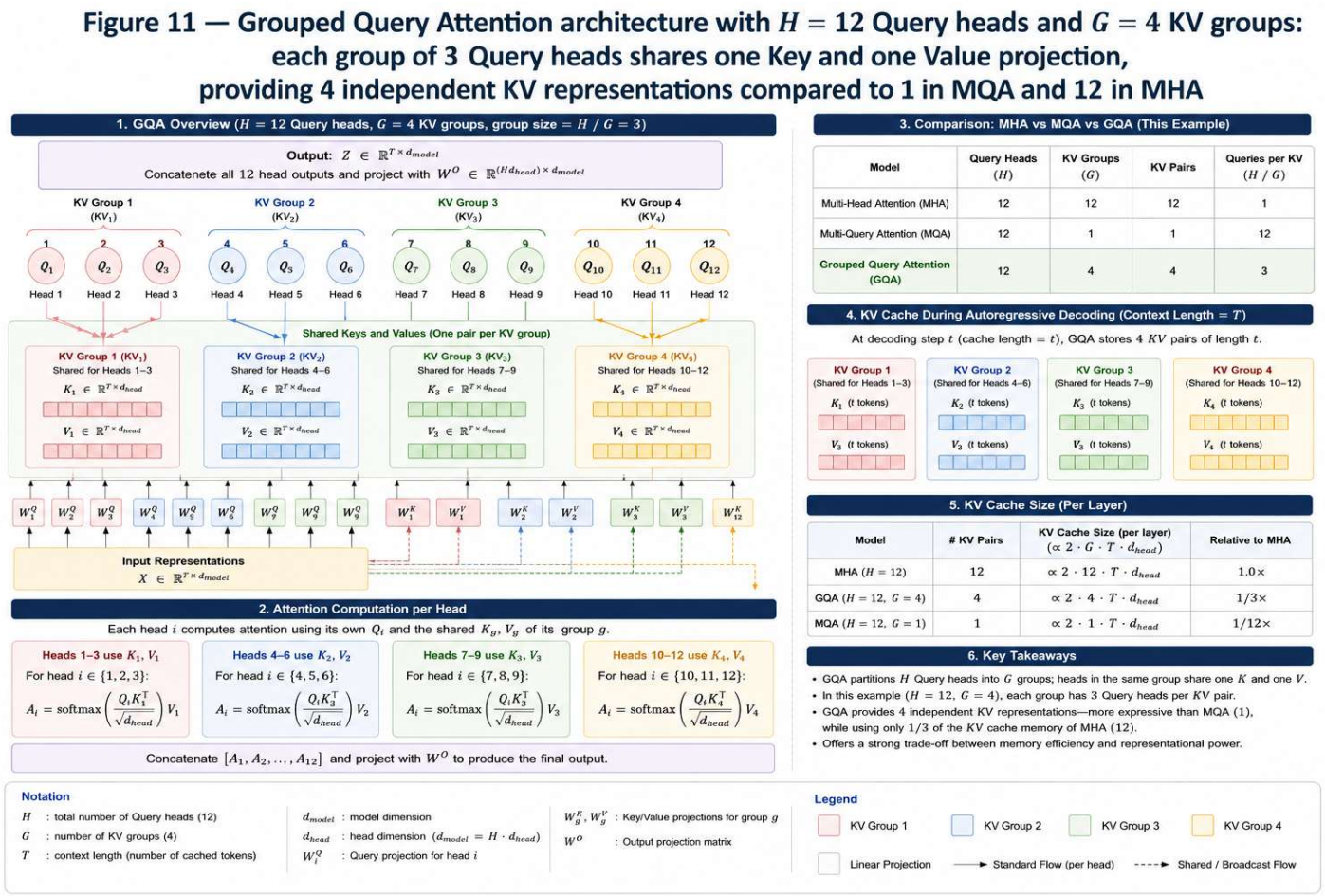



**Figure 11** — *Grouped Query Attention architecture with H = 12 Query heads and G = 4 KV groups: each group of 3 Query heads shares one Key and one Value projection, providing 4 independent KV representations compared to 1 in MQA and 12 in MHA*

## 4.5 Mathematical Formulation

Let H denote the number of Query heads and G the number of KV groups, where G divides H. Each Query head i retains its own Query projection. Each group g = 1, ..., G has its own shared Key and Value projections. The group to which head i belongs is determined by the mapping γ(i) = ⌈i × G/H⌉.

Query projection (unchanged from MHA, head-specific):

$$Q_i = X\,W_i^Q \tag{14}$$

Key and Value projections (per group, shared within group):

$$K_g = X\,W_g^K,\ \ g = 1, \ldots, G \tag{15}$$

$$V_g = X\,W_g^V,\ \ g = 1, \ldots, G \tag{16}$$

Attention computation for head i, using the Key and Value from its assigned group γ(i):

$$\text{Attention}_i = \text{softmax}(Q_i K_{\gamma(i)}^T / \sqrt{d_k}) \cdot V_{\gamma(i)} \tag{17}$$

The final output is formed by concatenating all H head outputs and projecting through the shared output matrix, unchanged from MHA and MQA:

$$\text{GQA}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\,W^O \tag{18}$$

The KV cache for Grouped Query Attention stores G Key tensors and G Value tensors per token per layer:

$$\text{KV Cache (GQA)} = 2 \times L \times G \times T \times d_h \times p \tag{19}$$

This equation interpolates exactly between the two extremes. When G = 1, Equation 19 reduces to Equation 13 (Multi-Query Attention). When G = H, it reduces to Equation 7 (Multi-Head Attention). The KV cache is G/H the size of MHA. For H = 32 and G = 4, the KV cache is 4/32 = 1/8 the size of MHA, while storing four times more than MQA.

**Table 5** — *Notation reference for Grouped Query Attention equations (14)–(19)*

| Symbol | Description |
|---|---|
| $G$ | Number of KV groups (1 ≤ G ≤ H); the key design parameter of GQA |
| $K_g$ | Shared Key tensor for group g; one per group per layer |
| $V_g$ | Shared Value tensor for group g; one per group per layer |
| $W_g^K$ | Key projection matrix for group g |
| $W_g^V$ | Value projection matrix for group g |

| | |
|---|---|
| γ(i) | Group assignment function for Query head i: ⌈i × G/H⌉ |
| H/G | Number of Query heads per group (group size) |
| G/H | KV cache as a fraction of MHA; reduction factor = H/G |

## 4.6 Autoregressive Decoding: A Worked Example

Continuing the 'Hi how are you' example with H = 12 Query heads and G = 4 KV groups (group size = 3 Query heads per group), the decoding process unfolds as follows.

***Step 1: Processing 'Hi'***

Each transformer layer computes 12 independent Query vectors $Q_1$, ..., $Q_{12}$ using 12 independent projection matrices. It also computes 4 Key vectors and 4 Value vectors—one Key-Value pair per group. The KV cache stores these 8 tensors (4 Keys + 4 Values) per layer. Query heads $Q_1$–$Q_3$ attend to $K_1$ and retrieve from $V_1$; heads $Q_4$–$Q_6$ attend to $K_2$ and retrieve from $V_2$; and so on. Heads within the same group use different Q vectors, so they still produce different attention distributions over the sequence, even though they read from the same Key and Value tensors.

***Step 2: Generating 'how'***

The model computes 12 new Query vectors and 4 new Key-Value pairs for 'how'. The new KV tensors are appended to each group cache: each of the 4 group caches now contains two Key-Value pairs. Every Query head attends over the two-token history using its group's cached Keys, and retrieves from its group's cached Values. The KV cache contains 4 × 2 = 8 Key tensors and 8 Value tensors across all groups.

***Steps 3–4: Generating 'are' and 'you'***

The pattern continues. After decoding all four tokens, the KV cache across all layers contains 4 × 4 = 16 Key tensors and 16 Value tensors per layer. Multi-Head Attention would store 12 × 4 = 48 Key and 48 Value tensors; Multi-Query Attention would store 1 × 4 = 4 of each. Grouped Query Attention stores three times more KV data than MQA but four times less than MHA for this configuration. The HBM reads per decoding step scale accordingly: larger than MQA but substantially smaller than MHA.

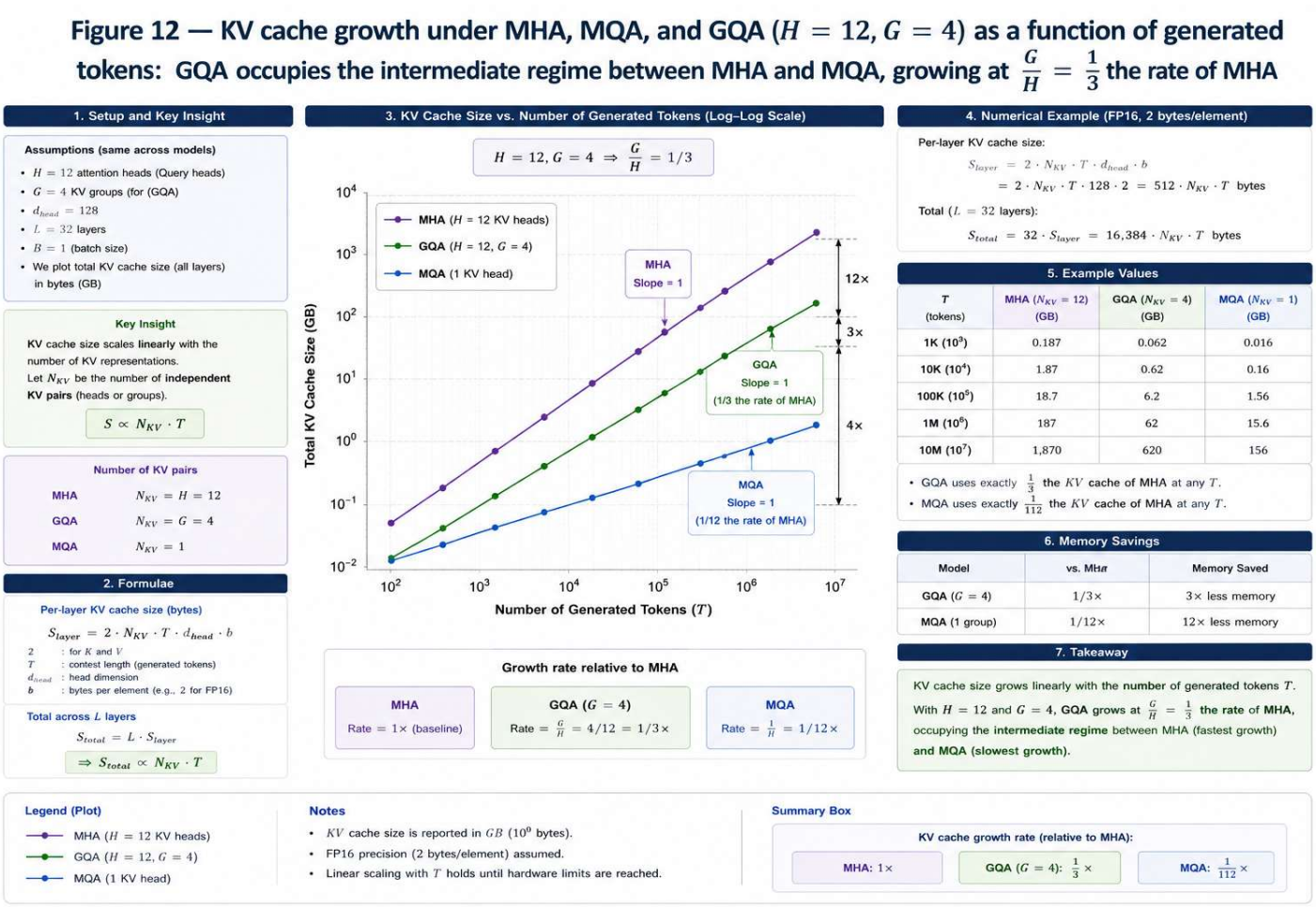



**Figure 12** — *KV cache growth under MHA, MQA, and GQA (H = 12, G = 4) as a function of generated tokens: GQA occupies the intermediate regime between MHA and MQA, growing at G/H = 1/3 the rate of MHA*

## 4.7 Why Accuracy Improves over Multi-Query Attention

The accuracy advantage of Grouped Query Attention over Multi-Query Attention follows directly from the structure of the grouping. To understand why, it is necessary to examine what is lost in Multi-Query Attention and how Grouped Query Attention partially restores it.

In Multi-Query Attention, the gradient signal flowing through the single shared Key projection during training is the aggregate of gradients from all H Query heads. Some heads develop syntactic specialisations, others semantic, others positional. The gradient from a syntactic head signals that the Key projection should emphasise structural features; the gradient from a semantic head signals that it should emphasise entity identity. These are conflicting requirements applied to a single parameter matrix. The result is a Key projection that is a compromise—capturing some structure but not precisely matching any head's optimal projection.

In Grouped Query Attention, each group g maintains its own Key projection $W^G K$. The gradient flowing through $W^G K$ is the aggregate only of the H/G Query heads assigned to group g. Adjacent heads in the attention head index space tend to develop related specialisations during training. With G = 4 groups of 3 heads each, each Key projection receives a more coherent gradient signal and can specialise more

precisely to the needs of its three associated Query heads. The representational compromise required is substantially smaller than in Multi-Query Attention.

An engineering analogy makes the intuition concrete. Consider twelve specialist engineers, each working on a different subsystem of a large software platform. In Multi-Query Attention, all twelve engineers are issued a single shared technical database. It must serve compiler engineers, networking engineers, storage engineers, security engineers, and graphics engineers simultaneously. Every engineer retrieves information from a database not curated for their domain.

In Grouped Query Attention with G = 4, engineers are divided into four teams by domain affinity: systems-level engineers share one database, application-level engineers share another, security engineers share a third, and performance engineers share a fourth. Each database is curated to the needs of its three-engineer team. Each engineer retrieves from a resource that more closely matches their domain. No engineer requires their own private database—the efficiency of sharing is preserved—but the shared resource is now tailored to a smaller, more homogeneous group.

The analogy maps directly to Query, Key, and Value. Each Query head is an engineer asking a domain-specific question (Q). The Key database determines which tokens are 'visible' as relevant to that question (K). The Value database determines what information is extracted from the relevant tokens (V). In GQA, each group receives a Key and Value curated to the needs of its Query heads. The specialisation is not as deep as full per-head independence (MHA), but it is substantially better than a single shared representation across all heads (MQA).

This intuition is supported by the empirical findings of Ainslie et al. (2023), who demonstrated that GQA models trained with G = H/4 to H/8 groups achieve perplexity and downstream task performance within one to two percent of Multi-Head Attention baselines, while retaining most of the inference efficiency gains of Multi-Query Attention. The quality recovery is most pronounced for tasks requiring diverse multi-step reasoning and long-range dependency tracking—precisely the tasks most sensitive to per-head representational diversity.

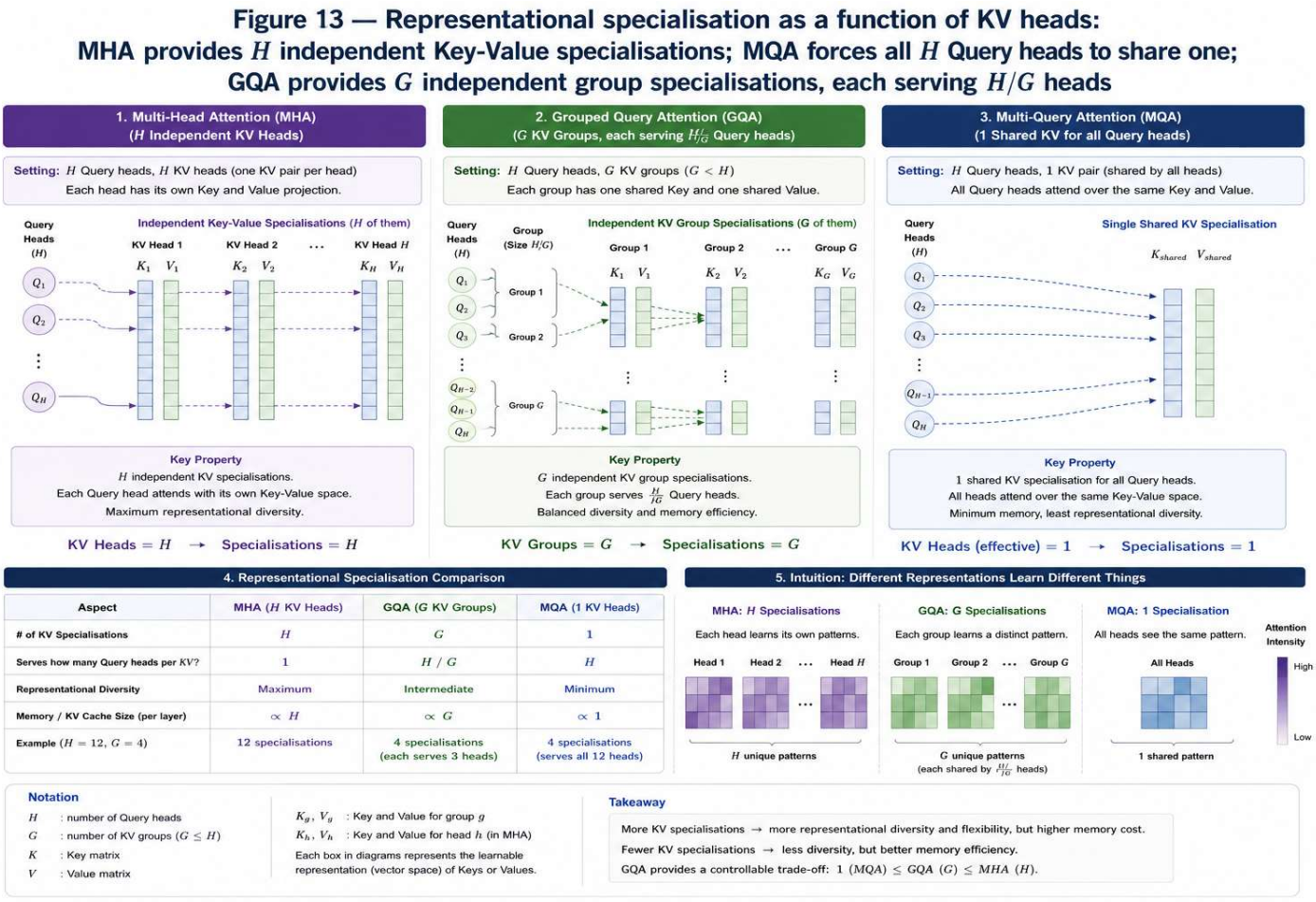



**Figure 13** — *Representational specialisation as a function of KV heads: MHA provides H independent Key-Value specialisations; MQA forces all H Query heads to share one; GQA provides G independent group specialisations, each serving H/G heads*

## 4.8 Engineering Trade-offs

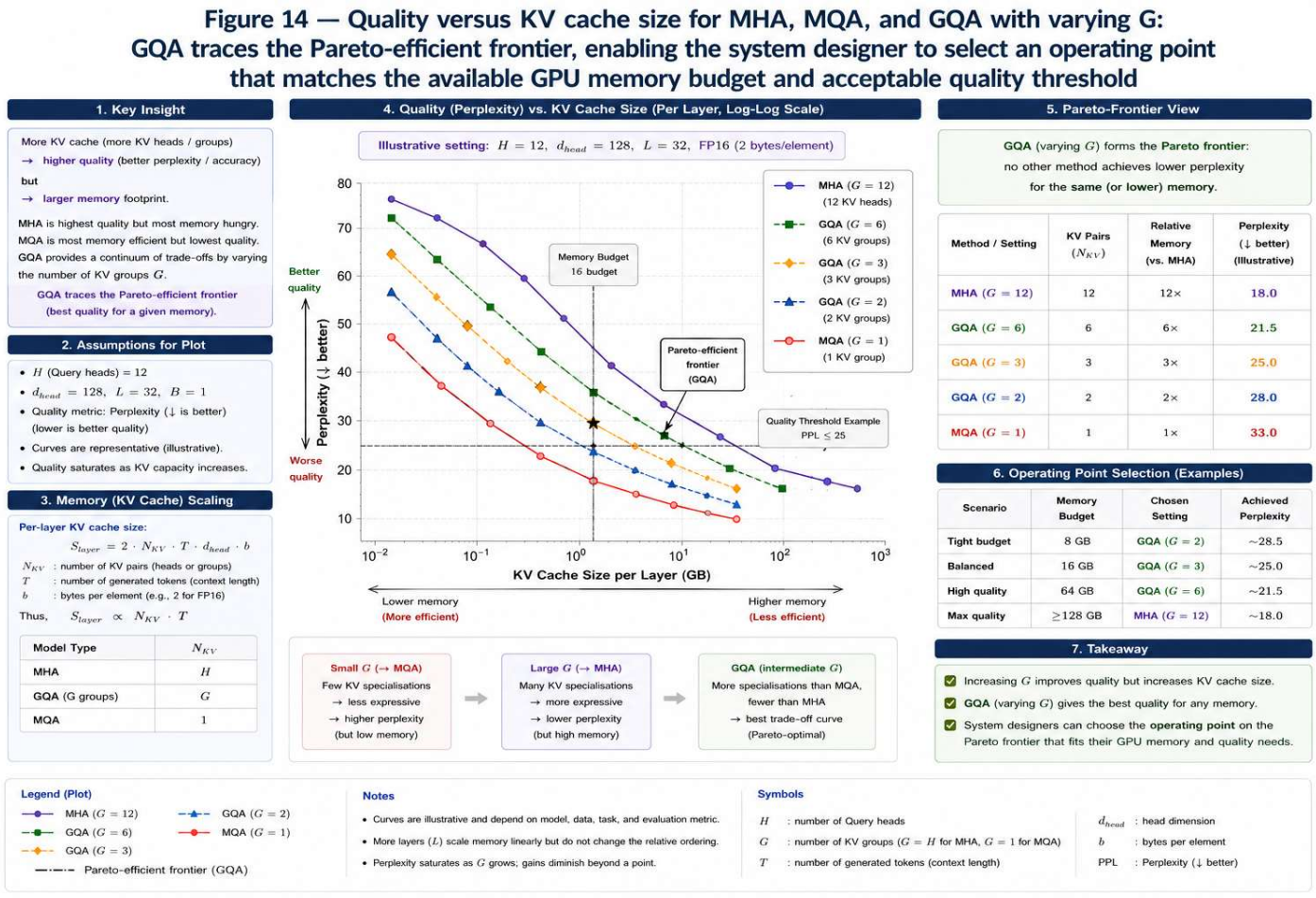
The following table compares Multi-Head Attention, Multi-Query Attention, and Grouped Query Attention across the dimensions most relevant to production inference system design.

**Table 6** — *Engineering comparison of MHA, MQA, and GQA across key inference and quality dimensions*

| Feature | Multi-Head Attention | Multi-Query Attention | Grouped Query Attention |
|---|---|---|---|
| Query heads | H (independent) | H (independent) | H (independent) |
| KV heads | H | 1 | G (tunable) |
| KV cache size | $2LHTd_hp$ (baseline) | $2L{\cdot}1{\cdot}Td_hp$ (1/H of MHA) | $2LGTd_hp$ (G/H of MHA) |

| | | | |
|---|---|---|---|
| HBM reads per step | Highest (H sets) | Lowest (1 set) | Intermediate (G sets) |
| Memory efficiency | Lowest | Highest | High (tunable via G) |
| Representational diversity | Maximum (per-head KV) | Minimum (single shared KV) | Intermediate (per-group KV) |
| Model quality | Highest | Measurably lower | Near-MHA for G ≥ H/8 |
| Inference latency | Highest | Lowest | Near-MQA for small G |
| Batch size capacity | Smallest | Largest | Large (tunable) |
| Industry adoption (2024+) | Legacy and specialised use | Limited deployment | Dominant standard |

**Figure 14 — Quality versus KV cache size for MHA, MQA, and GQA with varying G: GQA traces the Pareto-efficient frontier, enabling the system designer to select an operating point that matches the available GPU memory budget and acceptable quality threshold**



## 4.9 Why Modern LLMs Use Grouped Query Attention

Grouped Query Attention has become the standard attention mechanism across virtually every major openly-released language model architecture introduced since 2023. LLaMA 2 introduced GQA in its larger variants, using G = 8 KV heads across H = 64 Query heads for the 70B model. LLaMA 3 extended GQA to all model scales including the 8B model, using H = 32 Query heads and G = 8 KV heads. Mistral 7B adopted H = 32, G = 8, demonstrating inference speed competitive with much larger MQA models while maintaining quality competitive with or exceeding LLaMA 2 13B under standard benchmarks. Gemma, Qwen, and Phi model families similarly adopt GQA configurations, with the specific choice of G tuned to the target deployment environment and hardware constraints.

The industry convergence on GQA reflects a specific engineering judgment: the quality loss of Multi-Query Attention is too large to accept for general-purpose deployment, while the memory overhead of Multi-Head Attention is too costly at the scales and context lengths required by production workloads. Grouped Query Attention resolves this tension by providing a tunable interpolation. Several properties drive its adoption.

First, the quality–efficiency Pareto frontier is practical. With G = H/4 to H/8, GQA retains 95–98 percent of MHA quality in typical evaluation regimes while achieving 75–90 percent of MQA's KV cache reduction. This operating point is viable for deployment; neither the quality nor the efficiency sacrifice is prohibitive.

Second, the design is compatible with existing serving infrastructure. GQA attention kernels are structurally similar to MQA kernels, requiring only minor modifications to handle the group-to-head mapping. This reduces the engineering cost of adoption relative to more radical architectural changes.

Third, G is a tunable hyperparameter that allows the model designer to optimise for the target deployment hardware. A model intended for inference on 16 GiB GPUs can use a smaller G to fit within the memory budget; a model intended for 80 GiB GPUs can afford a larger G for higher quality. This flexibility is practically valuable.

GQA does not eliminate the KV cache bottleneck. It reduces it substantially. For context windows of 32,000 to 128,000 tokens—now routine in frontier deployments—even a GQA model with G = H/8 can accumulate several gigabytes of KV cache per request at frontier model scales. This motivates continued research into attention mechanisms that compress the Key and Value representations themselves, rather than merely reducing their number.

## 4.10 Lessons Learned: The Attention Efficiency Progression

The three mechanisms examined in Chapters 2 through 4 constitute a systematic progression in the engineering understanding of attention efficiency. Each architecture answers a more refined version of the same question: how should Key and Value information be organised to minimise inference memory cost while preserving model quality?

Multi-Head Attention establishes the representational upper bound. Every head maintains fully independent Key and Value projections, enabling maximal specialisation. The cost is a KV cache that scales as $2 \times L \times H \times T \times d_h \times p$ bytes—which at frontier model scales and long context windows represents the dominant constraint on inference throughput and concurrent serving capacity.

Multi-Query Attention establishes the efficiency lower bound. A single shared Key and Value serves all Query heads, reducing the KV cache by a factor of H. This demonstrates that Key and Value projections contain substantial redundancy across heads, and that this redundancy can be exploited without completely destroying model utility. The cost is a quality degradation that is measurable and, for general-purpose deployment, typically unacceptable.

Grouped Query Attention identifies the engineering optimum within this design space. A small number of group-specific Key and Value projections recovers most of the quality lost by Multi-Query Attention while retaining most of the efficiency gains. The group count G is a single hyperparameter that the system designer can tune to balance quality against memory budget, explaining its rapid adoption as the industry default.

## Chapter Summary

Grouped Query Attention occupies the practical engineering optimum within the design space defined by Multi-Head Attention and Multi-Query Attention. By partitioning H Query heads into G groups and assigning each group its own shared Key and Value projection, GQA provides G independent KV representations that can each specialise to the needs of a smaller subset of Query heads. This restores much of the representational diversity eliminated by Multi-Query Attention while preserving most of the inference efficiency gains.

The KV cache under GQA scales as $2 \times L \times G \times T \times d_h \times p$, a factor of H/G smaller than Multi-Head Attention. For typical production configurations with G = H/4 to H/8, this reduction is four to eight times. Empirically, this configuration achieves model quality within one to two percent of Multi-Head Attention baselines while closely approaching the inference efficiency of Multi-Query Attention. These properties have established GQA as the standard attention mechanism in modern production language models including LLaMA 3, Mistral, Gemma, and Qwen.

The progression from Multi-Head Attention through Multi-Query Attention to Grouped Query Attention illustrates a general principle in the engineering of neural network inference: representational redundancy can be systematically identified and reduced without proportional quality loss, provided the reduction is applied at an appropriate granularity. Multi-Query Attention reduced at too coarse a granularity; Grouped Query Attention identifies the appropriate balance.

Yet even Grouped Query Attention does not eliminate the KV cache problem. At context lengths of 64,000 to 128,000 tokens—now routine in frontier deployments—and at parameter scales of seventy billion or more, GQA models still accumulate tens of gigabytes of KV cache per inference session. The mechanisms examined thus far reduce the number of Key-Value heads while leaving the representations themselves unchanged. The following chapter examines a fundamentally different approach: compressing the Key and Value representations themselves into a low-dimensional latent space. Multi-Head Latent Attention, introduced in the DeepSeek-V2 architecture, achieves KV cache reductions that go beyond what KV head reduction alone can provide, and introduces a new set of engineering trade-offs that define the current frontier of attention mechanism design.

## 5. Multi-Head Latent Attention

The preceding three chapters traced the evolution of attention mechanisms through a sequence of architectural decisions, each motivated by the engineering bottleneck left unresolved by its predecessor. Multi-Head Attention established the representational standard. Multi-Query Attention demonstrated that Key and Value projections can be shared across heads without catastrophic quality loss. Grouped Query Attention refined this insight, recovering most of the quality gap while preserving most of the efficiency gain. By 2024, GQA had become the dominant attention mechanism in production language models. Yet the KV cache problem was not solved—it was merely deferred.

Multi-Head Latent Attention (MLA), introduced in the DeepSeek-V2 architecture (DeepSeek-AI, 2024), approaches the KV cache problem from a fundamentally different direction. Rather than asking how many Key and Value heads to retain, MLA asks whether the Key and Value representations themselves need to be stored at full dimensionality. The answer, it demonstrates, is no. By projecting Keys and Values into a compact low-dimensional latent space before caching, MLA achieves KV cache reductions that go beyond what any KV head reduction strategy can provide, while simultaneously achieving stronger model quality than Multi-Head Attention on standard evaluation benchmarks.

### 5.1 Why GQA Was Still Not Enough

Grouped Query Attention addressed the per-head redundancy in the KV cache by reducing the number of Key-Value heads from H to G. For a typical production configuration with G = H/8, this reduces the KV cache by a factor of eight relative to Multi-Head Attention. However, GQA does not change the fundamental scaling behaviour of the KV cache. Equation 19 showed that the GQA KV cache scales as $2 \times L \times G \times T \times d_h \times p$ bytes. This expression is linear in T—the sequence length. Reducing G by a constant factor reduces the cache by a constant factor but does not change the exponent. As T grows, the memory consumption grows in exact proportion.

For the context lengths that defined production deployments through 2023—8K to 32K tokens—this linear growth was manageable. GQA with G = H/8 kept the KV cache within practical bounds. However, context windows have continued to extend rapidly. Frontier models now routinely support 128K tokens; research models have demonstrated contexts of 1 million tokens or more. At these scales, even a dramatically reduced GQA configuration accumulates substantial KV cache.

To make the problem concrete: consider a model with L = 60 layers, G = 8 KV heads, $d_h = 128$, and FP16 precision. At T = 128,000 tokens, the GQA KV cache is $2 \times 60 \times 8 \times 128{,}000 \times 128 \times 2$ bytes = approximately 31.5 GiB per request. On an H100 GPU with 80 GiB of HBM, a single request at full context consumes nearly 40 percent of available GPU memory before accounting for model weights. At T = 1,000,000 tokens, the same configuration requires approximately 246 GiB—far exceeding any single GPU's capacity.

The underlying cause is that GQA, like MQA and MHA before it, stores complete Key and Value vectors for every token in the cache. The stored representation for each token has dimensionality $G \times 2 \times d_h$—and while G is smaller than H, every dimension of every KV vector is stored at full precision for every token. The linear growth with T is a consequence of this: adding one more token to the context always adds exactly $G \times 2 \times d_h \times p$ more bytes to the cache, regardless of how many tokens are already cached.

This observation points to a different class of optimisation. Previous attention mechanisms reduced the coefficient multiplying T in the cache formula by reducing G. But G cannot be reduced below 1 (which is Multi-Query Attention), and MQA's quality penalty is already deemed too large for general-purpose

deployment. No amount of further reduction in the number of KV heads can eliminate the linear scaling. A different approach is required.

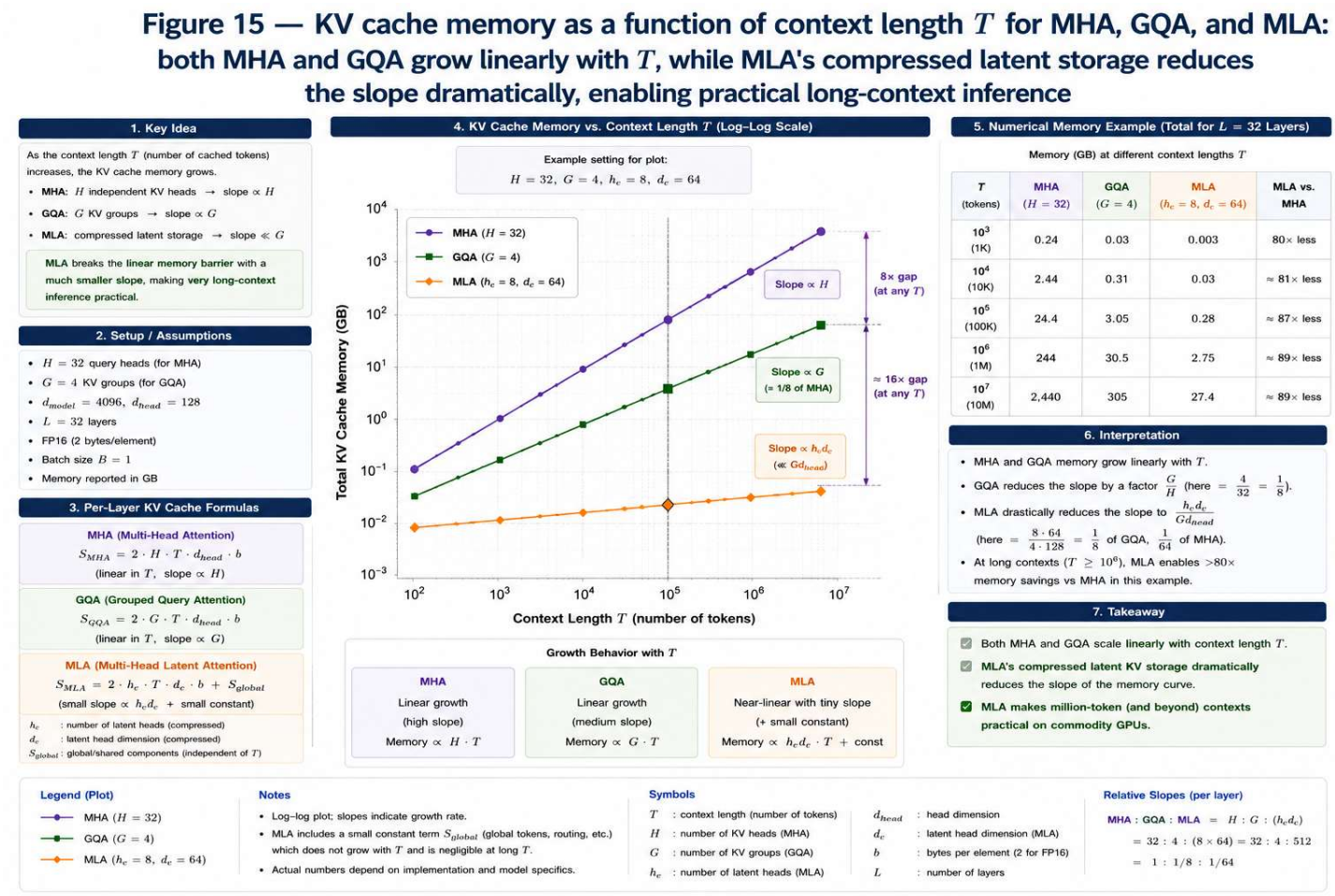



**Figure 15** — *KV cache memory as a function of context length T for MHA, GQA, and MLA: both MHA and GQA grow linearly with T, while MLA's compressed latent storage reduces the slope dramatically, enabling practical long-context inference*

## 5.2 A Fundamentally Different Engineering Question

Every attention mechanism examined in the preceding chapters—Multi-Head Attention, Multi-Query Attention, and Grouped Query Attention—approached the KV cache problem by asking a variant of the same question: how many sets of Key and Value representations must be stored? MHA stores H sets. MQA stores 1. GQA stores G, with 1 ≤ G ≤ H. The design variable is the number of KV heads.

DeepSeek-AI (2024) recognised that this entire family of approaches leaves a dimension of the problem unexplored: the dimensionality of each stored representation. Every approach in the MHA-MQA-GQA family stores Key and Value vectors at full head dimension $d_h$. This is natural—Keys and Values are generated at $d_h$ dimensions and used at $d_h$ dimensions—but it is not necessary. If the information required to reconstruct Key and Value vectors can be represented in a lower-dimensional space, then only that compact representation needs to be cached. The full Key and Value tensors can be reconstructed from this compact representation at inference time, when they are needed for the attention computation.

This is the central insight of Multi-Head Latent Attention: the Key and Value representations for all heads can be jointly compressed into a shared low-dimensional latent vector, and this latent vector alone is what needs to be stored in the KV cache. The question shifts from 'how many KV heads?' to 'how many dimensions are actually necessary to faithfully represent the KV information?' The answer, empirically, is far fewer than n_h × $d_h$.

This is conceptually analogous to lossy compression in signal processing. A high-resolution image contains more information than is necessary to reconstruct a perceptually faithful representation. A JPEG encoder identifies a compact representation that captures the essential information; the decoder reconstructs an approximation of the original from this compact form. MLA performs an analogous operation on the Key and Value representations: a learned encoder (the down-projection matrix) compresses the KV information into a compact latent vector; a learned decoder (the up-projection matrices) reconstructs the necessary Key and Value tensors at query time. Unlike JPEG, both the encoder and decoder are learned jointly end-to-end through training on the language modelling objective, meaning the compression is optimised specifically for the task rather than for generic perceptual fidelity.

## 5.3 Multi-Head Latent Attention Architecture

Multi-Head Latent Attention uses low-rank joint compression to produce a compact representation of the Key and Value information for every token. Let h_t ∈ $\mathbb{R}^d$ denote the attention input of the t-th token, where d is the model dimension. The MLA mechanism proceeds in three stages: compression into a latent vector, caching only the latent vector, and reconstruction at inference time.

### *Stage 1: KV Compression (Down-Projection)*

For each token t, a single down-projection matrix W^DKV ∈ ℝ^(d_c × d) compresses the attention input h_t into a compact latent vector:

$$c^{tKV} = W^{DKV} h_t \tag{20}$$

where c_t^KV ∈ ℝ^(d_c) is the compressed latent vector for keys and values, d_c is the KV compression dimension, and d_c ≪ n_h × d_h. This latent vector is what is stored in the KV cache. For DeepSeek-V2, d_c = 512, n_h = 128, and d_h = 128, so the latent vector has 512 dimensions compared to the 2 × 128 × 128 = 32,768 elements that full MHA would cache per token.

### *Stage 2: KV Reconstruction (Up-Projection)*

When attention must be computed for a new token, the cached latent vectors c_j^KV for all previous tokens j = 1, ..., t are retrieved from HBM. The Key and Value tensors are then reconstructed using learned up-projection matrices W^UK ∈ ℝ^(n_h × d_h × d_c) and W^UV ∈ ℝ^(n_h × d_h × d_c):

$$k_t^C = W^{UK} c_t^{KV} \tag{21}$$

$$v_t^C = W^{UV} c_t^{KV} \tag{22}$$

Here k_t^C ∈ ℝ^(n_h × d_h) and v_t^C ∈ ℝ^(n_h × d_h) are the reconstructed content Key and Value tensors. The superscript C denotes 'content', distinguishing these from the position-encoding component introduced in Section 5.3.3 below.

***Stage 3: Matrix Absorption — Inference Without Explicit Reconstruction***

A critical engineering optimisation follows from the structure of Equations 21 and 22. The query vector Q_t is produced by projecting h_t through the Query weight matrix W^Q: Q_t = W^Q × h_t. The attention score between query t and the content key of token j is then Q_t^T × k_j^C = (W^Q × h_t)^T × (W^UK × c_j^KV). Since this can be rewritten as h_t^T × (W^Q)^T × W^UK × c_j^KV, the product (W^Q)^T × W^UK can be precomputed once and treated as a single merged weight matrix. During inference, the attention score can therefore be computed directly by multiplying the query against the cached latent vector c_j^KV, without ever explicitly computing the full Key tensor k_j^C. An analogous absorption applies to W^UV into W^O for the Value side.

This absorption trick means that, in practice, the reconstructed Key and Value tensors never materialise in GPU memory during inference. The attention computation operates directly on the compact latent representation. This further reduces memory pressure during the attention computation itself, beyond the savings already achieved by storing only the latent vector in the KV cache.

### *5.3.4 Decoupled Rotary Position Embedding*

Rotary Position Embedding (RoPE) is the dominant position encoding strategy in modern LLMs. It encodes positional information by applying a rotation to the Query and Key vectors before computing attention scores, where the rotation angle depends on the token position. This is incompatible with the matrix absorption trick described above.

The incompatibility arises because RoPE applies a position-dependent rotation matrix to each Key vector. If RoPE is applied to k_t^C = W^UK × c_t^KV, the result is RoPE(W^UK × c_t^KV). This position-sensitive term cannot be absorbed into W^Q during inference, because a different RoPE rotation matrix exists at every token position j, and matrix multiplication does not commute with a position-varying rotation. As a result, if RoPE were applied to the content Keys, the Keys for all prefix tokens would need to be recomputed at every decoding step—defeating the purpose of the KV cache.

DeepSeek-AI (2024) resolves this with a decoupled RoPE strategy. Position information is carried by a separate set of decoupled Query and Key vectors, computed independently from the latent compression pathway:

$$q_t^R = \mathrm{RoPE}(W^{QR} c_t^Q) \tag{23}$$

$$k_t^R = \mathrm{RoPE}(W^{KR} h_t) \tag{24}$$

where c_t^Q is the compressed Query latent vector (computed analogously to c_t^KV for training memory efficiency), W^QR ∈ ℝ^(d_h^R × n_h × d'_c) produces n_h decoupled Query vectors of dimension d_h^R, and W^KR ∈ ℝ^(d_h^R × d) produces a single shared decoupled Key of dimension d_h^R. The full Query and Key for head i are then formed by concatenating content and decoupled components:

$$q_{t,i} = [q_{t,i}^{C} ; q_{t,i}^{R}] \tag{25}$$

$$k_{t,i} = [k_{t,i}^{C} ; k_{t}^{R}] \tag{26}$$

The content components k_t,i^C are produced from the cached latent c_t^KV and do not carry position information (so W^UK can be absorbed). The decoupled component k_t^R carries RoPE position encoding and is shared across all heads—only one d_h^R-dimensional vector per token, not n_h. Both c_t^KV (content latent, d_c dimensions) and k_t^R (decoupled position key, d_h^R dimensions) must be stored in the KV cache. The total KV cache per token is therefore:

$$\text{KV Cache (MLA)} = (d_c + d_h^R) \times l \text{ elements per token} \tag{27}$$

For DeepSeek-V2 with d_c = 512 and d_h^R = 64, this is 576 elements per token per layer, compared to 2 × 128 × 128 = 32,768 elements for full MHA with n_h = 128 heads—a reduction to approximately 1.8 percent of the MHA cache, or roughly the equivalent of GQA with just 2.25 KV groups.

**Table 7** — *Notation reference for Multi-Head Latent Attention equations (20)–(27)*

| Symbol | Dimensions | Description |
|---|---|---|
| $h_t$ | $\mathbb{R}^d$ | Attention input (hidden state) of token t |
| $c_t^{KV}$ | $\mathbb{R}$^$d_o$ | Compressed KV latent vector — what is stored in the cache |
| $d_c$ | scalar | KV compression dimension ($d_c \ll n_h \times d_h$) |
| $W^{DKV}$ | $\mathbb{R}$^($d_o$ × d) | Down-projection matrix for KV compression |
| $W^{UK}$ | $\mathbb{R}$^($n_h \cdot d_h \times d_o$) | Up-projection matrix to reconstruct Keys |
| $W^{UV}$ | $\mathbb{R}$^($n_h \cdot d_h \times d_o$) | Up-projection matrix to reconstruct Values |
| $k_t^C$ | $\mathbb{R}$^(n_h × d_h) | Reconstructed content Key (not stored; absorbed or recomputed) |
| $v_t^C$ | $\mathbb{R}$^(n_h × d_h) | Reconstructed content Value (not stored; absorbed or recomputed) |
| $k_t^R$ | $\mathbb{R}$^(d_h^R) | Decoupled shared key carrying RoPE — stored in cache |
| $d_h^R$ | scalar | Dimension of decoupled RoPE keys/queries |
| RoPE(·) | — | Rotary position embedding rotation applied to a vector |

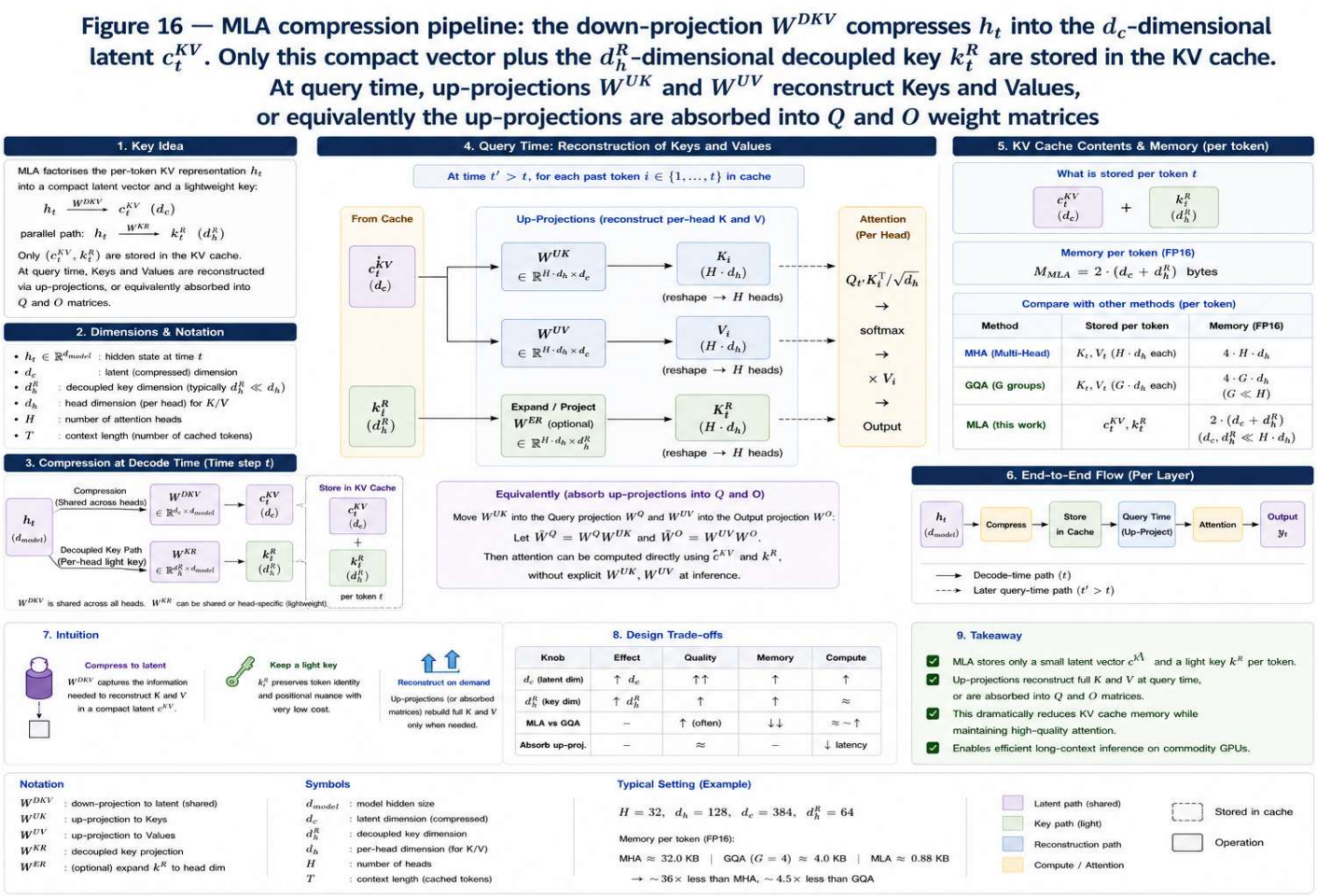



**Figure 16** — *MLA compression pipeline: the down-projection W^DKV compresses h_t into the d_c-dimensional latent c_t^KV. Only this compact vector plus the d_h^R-dimensional decoupled key k_t^R are stored in the KV cache. At query time, up-projections W^UK and W^UV reconstruct Keys and Values, or equivalently the up-projections are absorbed into Q and O weight matrices*

## 5.4 Autoregressive Decoding Under Multi-Head Latent Attention

Continuing the 'Hi how are you' example to illustrate MLA decoding under the DeepSeek-V2 configuration (n_h = 128, d_h = 128, d_c = 512, d_h^R = 64), the key difference from all previous mechanisms is what gets stored at each step.

### *Step 1: Processing 'Hi'*

The hidden state $h_1 \in \mathbb{R}^{5120}$ (model dimension 5,120 for DeepSeek-V2) is computed from the embedding of 'Hi' by the transformer layers up to and including the current one. The down-projection W^DKV ∈ ℝ^(512 × 5120) compresses $h_1$ into the latent vector $c_1$^KV ∈ ℝ^512. Simultaneously, the decoupled Key projection produces $k_1$^R = RoPE(W^KR × $h_1$) ∈ ℝ^64 carrying the position-1 encoding.

The KV cache stores ($c_1$^KV, $k_1$^R): 512 + 64 = 576 floating-point values per layer. No full Key or Value tensor is stored. The 128 × 128 = 16,384 Key dimensions and 128 × 128 = 16,384 Value dimensions that MHA

would store are never written to HBM. The 128 Query vectors (one per head) are computed, used for the attention computation over 'Hi', and then discarded.

***Step 2: Generating 'how'***

The hidden state $h_2$ for the query token 'how' is computed. For each head i, the Query vector q_{2,i} = [q_{2,i}^C ; q_{2,i}^R] is formed: the content component q_{2,i}^C comes from the compressed Query latent $c_2$^Q, and the decoupled component q_{2,i}^R carries the position-2 RoPE encoding. To compute attention over the prefix ['Hi'], the cached latent $c_1$^KV is retrieved from HBM and the Key k_{1,i}^C is reconstructed via W^UK × $c_1$^KV. The decoupled Key $k_1$^R is retrieved directly. The full Key k_{1,i} = [k_{1,i}^C ; $k_1$^R] and the attention score q_{2,i}^T × k_{1,i} are then computed. In practice, the matrix absorption trick means only $c_1$^KV needs to be loaded from HBM; the explicit reconstruction via W^UK is folded into the query-side computation.

The new latent $c_2$^KV and decoupled key $k_2$^R are computed from $h_2$ and appended to the KV cache. The cache now contains two 576-element entries. No full Key or Value history is stored.

***Steps 3–4: Generating 'are' and 'you'***

The pattern continues. After decoding all four tokens, the KV cache contains four 576-element entries per layer. At T = 4 tokens, this is 4 × 576 × 60 = 138,240 elements across all layers, compared to 4 × 32,768 × 60 = 7,864,320 elements for full MHA with the same architecture. The MLA cache is 56.8 times smaller for this example. As T grows, the ratio remains constant—the absolute reduction factor is determined by (d_c + d_h^R) / (2 × n_h × d_h), which for DeepSeek-V2 is 576 / 32,768 ≈ 1.76%.

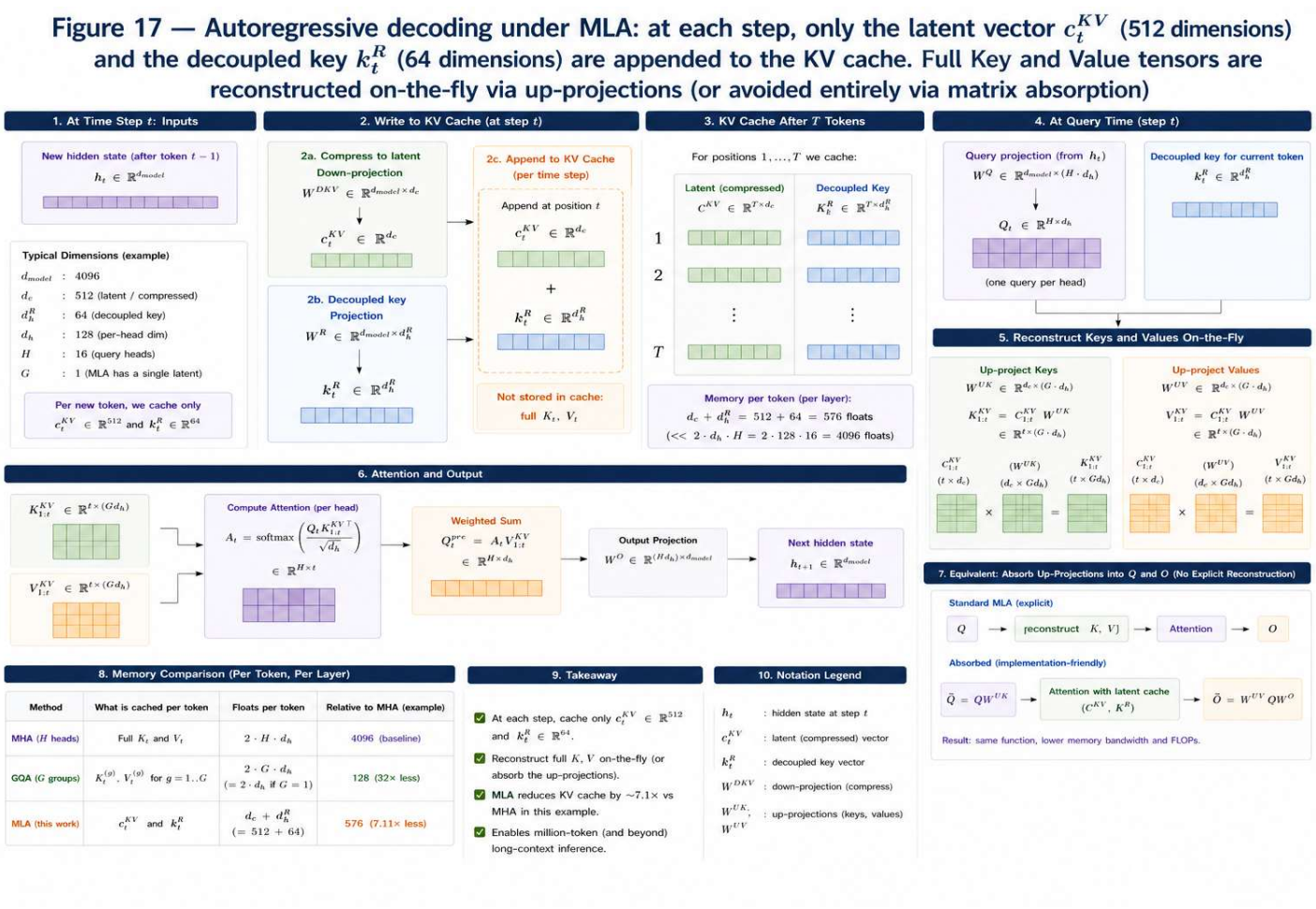



**Figure 17** — *Autoregressive decoding under MLA: at each step, only the latent vector c_t^KV (512 dimensions) and the decoupled key k_t^R (64 dimensions) are appended to the KV cache. Full Key and Value tensors are reconstructed on-the-fly via up-projections (or avoided entirely via matrix absorption)*

## 5.5 Memory Analysis

The KV cache memory requirements for all four attention mechanisms can now be compared precisely. The following expressions give the total KV cache in bytes for a sequence of T tokens across L layers at precision p bytes per element.

$$\text{KV Cache (MHA)} = 2 \times n_h \times d_h \times L \times T \times p \tag{28}$$

$$\text{KV Cache (MQA)} = 2 \times d_h \times L \times T \times p \tag{29}$$

$$\text{KV Cache (GQA)} = 2 \times G \times d_h \times L \times T \times p \tag{30}$$

$$\text{KV Cache (MLA)} = (d_c + d_h^R) \times L \times T \times p \tag{31}$$

All four expressions are linear in T—MLA does not change the asymptotic scaling class. However, the constant factor for MLA is dramatically smaller. For the DeepSeek-V2 architecture (n_h = 128, d_h = 128, d_c = 512, d_h^R = 64, L = 60, FP16):

**Table 8** — *KV cache memory per token (all layers) for DeepSeek-V2 architecture at FP16 (n_h = 128, d_h = 128, G = 8 for GQA, d_c = 512, d_h^R = 64, L = 60)*

| Mechanism | Elements per Token (all layers) | Bytes per Token (FP16) | KV Cache at T = 128K |
|---|---|---|---|
| Multi-Head Attention | 2 × 128 × 128 × 60 = 1,966,080 | ≈3.75 GiB | ≈480 TiB (impractical) |
| Multi-Query Attention | 2 × 1 × 128 × 60 = 15,360 | ≈29.3 MiB | ≈3.66 TiB (impractical) |
| Grouped Query Attention (G=8) | 2 × 8 × 128 × 60 = 122,880 | ≈234 MiB | ≈29.3 GiB |
| Multi-Head Latent Attention | (512 + 64) × 60 = 34,560 | ≈65.9 MiB | ≈8.24 GiB |

At 128K tokens with GQA (G = 8), the KV cache requires approximately 29.3 GiB per request—more than a third of an H100's 80 GiB HBM even before model weights. MLA reduces this to approximately 8.24 GiB, a factor of 3.6 improvement over GQA and a factor of approximately 58 over MHA. DeepSeek-AI (2024) report a 93.3% KV cache reduction relative to their previous 67B MHA-based model and a maximum generation throughput improvement of 5.76× using MLA on eight H800 GPUs.

Critically, MLA achieves this while performing better than MHA on standard language modelling and downstream task evaluations. This is not a quality-for-efficiency trade-off in the conventional sense: MLA does not sacrifice model quality to achieve memory savings. The low-rank compression of Keys and Values may in fact act as a regulariser during training, producing learned representations that generalise more effectively. The DeepSeek-V2 paper describes MLA as achieving 'stronger performance than MHA' while requiring a 'significantly smaller KV cache'.

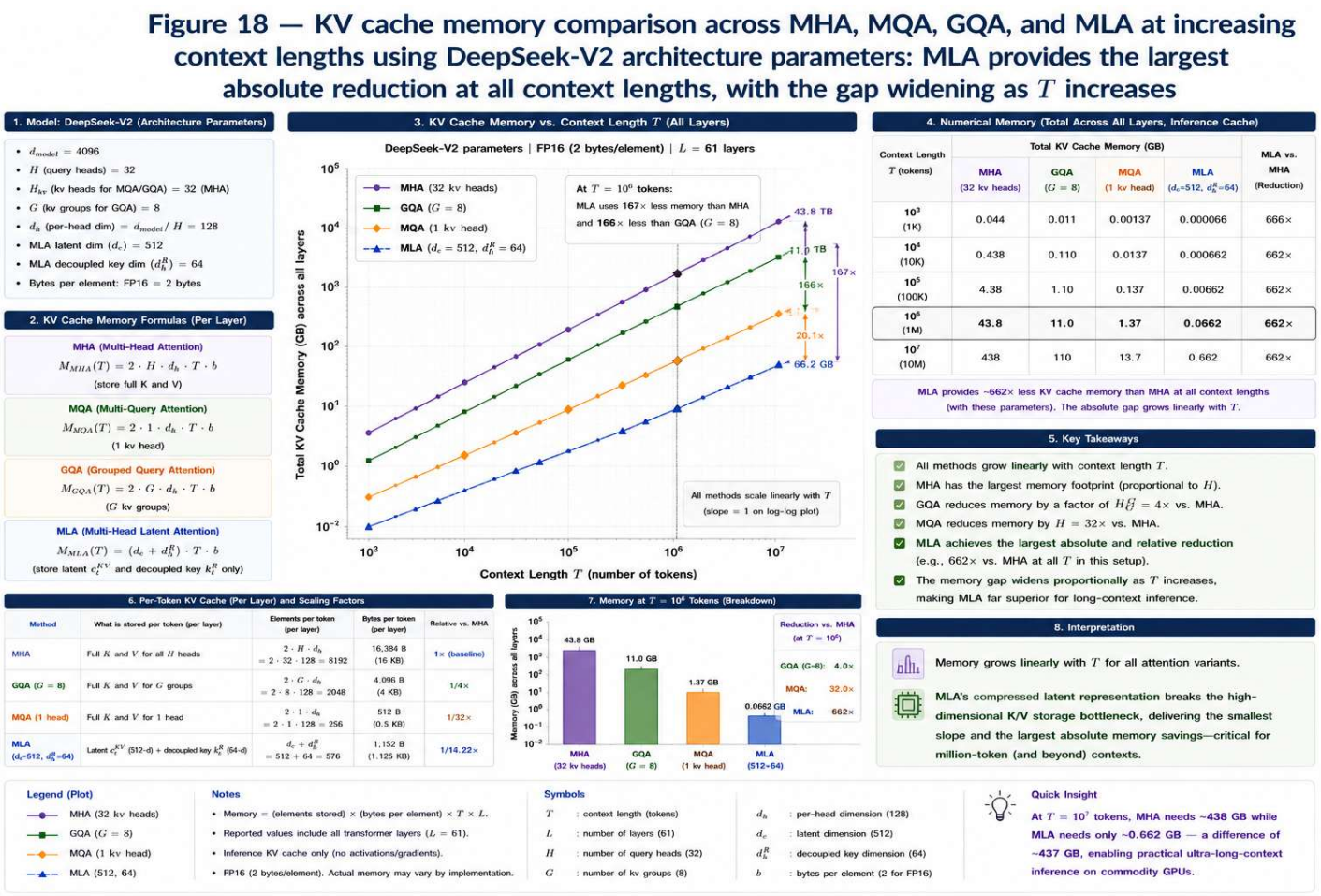



**Figure 18** — *KV cache memory comparison across MHA, MQA, GQA, and MLA at increasing context lengths using DeepSeek-V2 architecture parameters: MLA provides the largest absolute reduction at all context lengths, with the gap widening as T increases*

## 5.6 Engineering Trade-offs

Multi-Head Latent Attention offers a compelling set of engineering advantages over all preceding attention mechanisms, but it also introduces challenges that are absent from the simpler GQA design. Both aspects must be understood for a complete engineering assessment.

### *Advantages*

The primary advantage is dramatic KV cache reduction. For DeepSeek-V2, the reduction is 93.3% relative to the equivalent MHA model. This has direct consequences for three critical inference metrics. First, GPU memory availability: with substantially less memory consumed per request by the KV cache, more of the GPU's HBM can be allocated to model weights or to serving larger batch sizes. Second, HBM bandwidth: since the KV cache is smaller, each decoding step reads less data from HBM, reducing the bandwidth bottleneck identified in Chapter 2. Third, maximum batch size: more inference requests can be served concurrently within the same GPU memory budget, improving throughput in production serving scenarios.

A secondary advantage is that MLA does not reduce the number of Query heads. Unlike MQA and even GQA, the full set of n_h independent Query heads is preserved. Each head retains its own Query projection, and the attention distribution for each head is computed independently. The compression applies only to what is stored in the cache, not to the number of distinct attention computations performed at each step. This is why MLA can achieve stronger model quality than MHA despite a dramatically smaller cache: the representational capacity of the Query side is not constrained.

A third advantage is long-context scalability. Because the cache per token is smaller by a constant factor, MLA enables practical inference at context lengths that are infeasible for GQA. The constant factor reduction allows the same GPU memory budget to accommodate a proportionally longer context window, directly enabling applications such as long-document reasoning, extended dialogue, and code understanding over large repositories.

***Challenges***

The principal challenge of MLA is architectural complexity. The decoupled RoPE strategy, the down- and up-projection matrices, and the matrix absorption trick each introduce engineering complexity that is absent from GQA. Implementing an efficient MLA inference kernel requires careful attention to how the matrix absorptions interact with standard FlashAttention implementations. The DeepSeek-AI team reports implementing MLA on top of an improved version of FlashAttention-2, requiring non-trivial kernel engineering to achieve the claimed throughput.

A second challenge is that MLA is not a drop-in replacement for GQA. Models trained with GQA learn projection matrices whose shapes and semantics differ fundamentally from MLA's down- and up-projection structure. A GQA checkpoint cannot be used directly with an MLA attention kernel. Converting an existing GQA model to use MLA requires either full retraining—which is prohibitively expensive for frontier models—or a specialised uptraining procedure, discussed in Section 5.7.

A third consideration is the reconstruction compute cost. Although the matrix absorption trick eliminates the explicit materialisation of K and V in the common case, the down-projection (Eq. 20) must still be computed at prefill time for every token, and the absorbed projections add to the per-token FLOP count at decoding time. For highly compute-bound workloads, this additional compute may partially offset the bandwidth savings. In memory-bandwidth-bound inference scenarios—which represent the dominant operating regime for autoregressive decoding at practical batch sizes—the bandwidth savings dominate and the net effect is positive.

## 5.7 Uptraining Existing GQA Models with MLA

The vast majority of deployed decoder-only language models—including all members of the LLaMA, Mistral, Qwen, and Gemma families—are trained with Grouped Query Attention. Their learned weights are shaped by the GQA attention architecture: the Key and Value projection matrices have dimensions $n_g \times d_h \times d$ for $n_g$ GQA groups, not the low-rank down-projection structure of MLA. Adopting MLA in these models cannot be achieved by modifying a configuration file. The attention projection weights are structurally incompatible.

The question of whether an existing GQA model can be efficiently converted to MLA is therefore practically important. Full retraining of a frontier model from scratch with MLA built in—as DeepSeek-AI did for

DeepSeek-V2—is not a viable path for most organisations due to the cost of pretraining at scale. The TransMLA framework (Zeng et al., 2025; arXiv:2502.07864) addresses this problem directly.

TransMLA proposes an uptraining procedure that converts a pretrained GQA model into an MLA model. The key insight is that any GQA model's Key and Value weight matrices can be initialised as a rank-decomposed approximation of the original weights: the GQA Key projection W^K_g can be decomposed as W^UK × W^DKV, where the product approximates the original projection. This initialisation provides the new MLA model with a starting point that approximately preserves the pretrained model's behaviour, after which continued training on a relatively small amount of data recovers and exceeds the original performance.

Empirically, TransMLA demonstrates that fine-tuning on approximately 6 billion tokens—a small fraction of the hundreds of billions or trillions of tokens used for original pretraining—is sufficient to recover comparable benchmark performance across standard evaluation suites including reasoning, coding, and language understanding. The resulting model is architecturally identical to a natively trained MLA model and is fully compatible with DeepSeek's inference infrastructure, including SGLang and vLLM backends optimised for the MLA attention kernel.

The engineering significance of this result is substantial. It implies that the existing ecosystem of pretrained GQA models—representing enormous amounts of compute investment—can be adapted to use MLA without retraining from scratch. The uptraining procedure provides a practical migration path from the current GQA standard to MLA, enabling models trained before MLA was developed to benefit from MLA's inference efficiency improvements at modest additional cost.

It should be noted that the precise training requirements vary depending on model scale, base architecture, and target task distribution. The 6B token figure represents one empirical data point from the TransMLA study (Zeng et al., 2025) and should not be extrapolated without reference to the original paper and associated ablations. The claim that uptraining at approximately this scale recovers comparable performance is specific to the experimental conditions described therein.

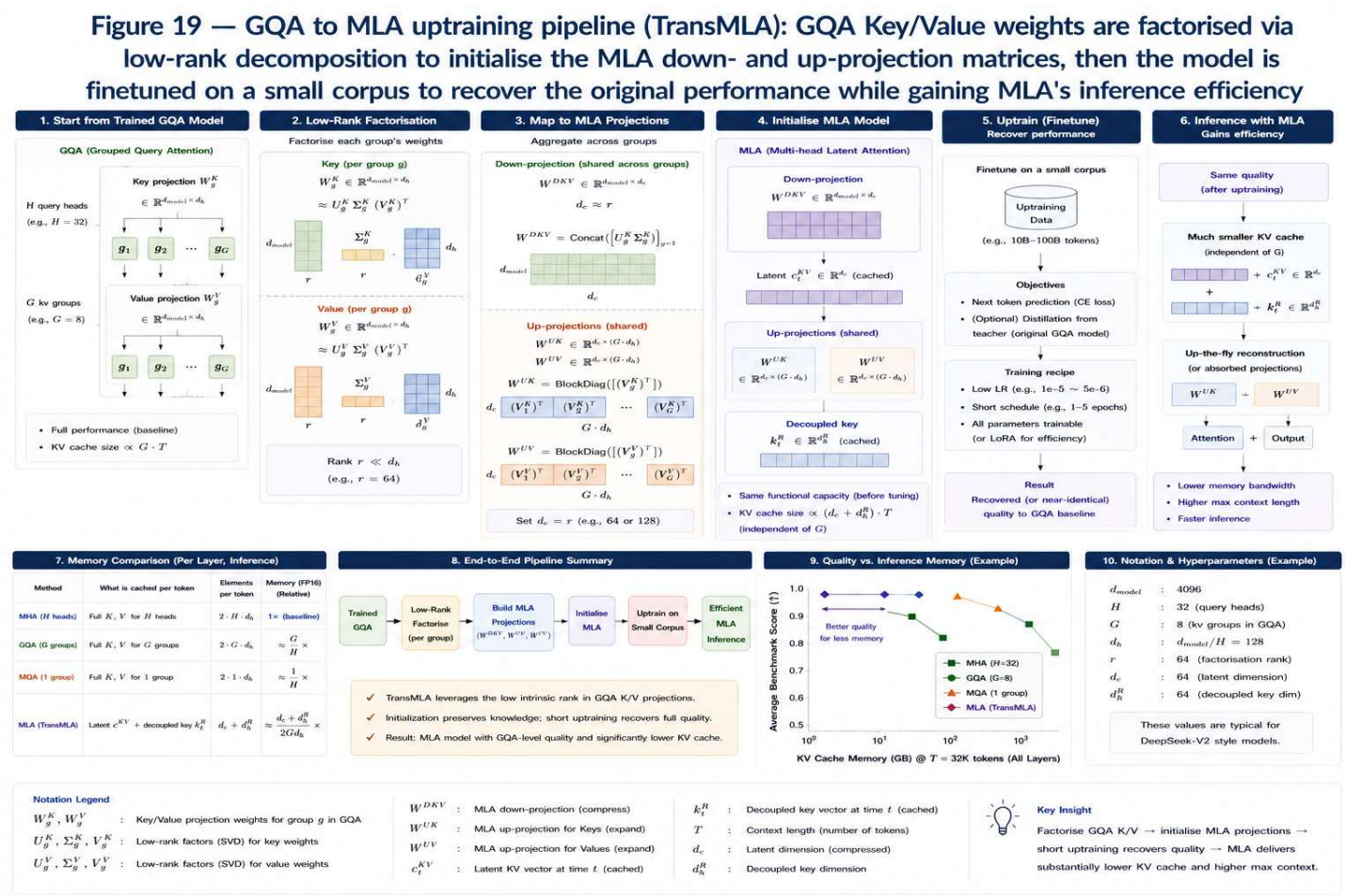



**Figure 19** — *GQA to MLA uptraining pipeline (TransMLA): GQA Key/Value weights are factorised via low-rank decomposition to initialise the MLA down- and up-projection matrices, then the model is finetuned on a small corpus to recover the original performance while gaining MLA's inference efficiency*

## 5.8 Comprehensive Architecture Comparison

The following table consolidates the engineering characteristics of all four attention mechanisms examined in this report across the dimensions most relevant to production inference system design.

**Table 9** — *Engineering comparison of MHA, MQA, GQA, and MLA across key architectural, memory, quality, and deployment dimensions*

| Feature | MHA | MQA | GQA | MLA |
|---|---|---|---|---|
| Query heads | n_h (independent) | n_h (independent) | n_h (independent) | n_h (independent) |
| KV heads | n_h | 1 | G (1 < G < n_h) | n_h (via up-proj) |
| Stored KV representation | Full K and V per head | One shared K, V | G shared K, V pairs | Compressed latent c^KV + decoupled k^R |

| | | | | |
|---|---|---|---|---|
| KV cache size (elements/token/layer) | 2n_h·d_h | 2d_h | 2G·d_h | d_c + d_h^R |
| KV cache vs MHA | 1× (baseline) | 1/n_h | G/n_h | (d_c+d_h^R)/(2n_h·d_h) |
| Example: DeepSeek-V2 scale | 32,768 elements | 256 elements | 2,048 (G=8) | 576 elements |
| HBM reads per decoding step | Highest | Lowest per token | Proportional to G | Lowest overall |
| Model quality | Strong | Weaker | Near-MHA | Stronger than MHA |
| Long-context (>128K) viability | Impractical | More viable | Challenging | Practical |
| Position encoding | Standard RoPE | Standard RoPE | Standard RoPE | Decoupled RoPE required |
| Inference kernel complexity | Standard | Simple | Simple | High (absorption, decoupled RoPE) |
| Requires training from scratch for adoption | N/A | Or uptraining | Or uptraining | Or TransMLA uptraining (~6B tokens) |
| Industry adoption (as of 2025) | Legacy large models | Niche/limited | Dominant standard | DeepSeek-V2, V3, R1; growing |

## Chapter Summary

Multi-Head Latent Attention represents a qualitatively different class of attention optimisation from its predecessors. Multi-Head Attention, Multi-Query Attention, and Grouped Query Attention all operate by varying the number of Key-Value heads stored in the cache. In doing so, they leave the dimensionality of each cached representation unchanged—every approach stores complete Key and Value vectors, differing only in how many.

MLA abandons this constraint. By projecting the joint Key-Value information into a compact low-dimensional latent vector via a learned down-projection, MLA caches representations that are orders of magnitude smaller than full Key-Value tensors. The full-dimensional Keys and Values are never stored; they are reconstructed on demand from the latent representation, or—via the matrix absorption trick—are never explicitly materialised at all during inference.

The engineering consequences are significant. For the DeepSeek-V2 architecture, MLA reduces the KV cache to approximately 1.76 percent of the MHA baseline—a 57-fold reduction—while achieving stronger model quality than MHA on standard benchmarks. At 128K context, the MLA KV cache fits within approximately 8.24 GiB per request on the same architecture where MHA would require over 480 TiB. This is not a marginal improvement. It is a categorical change in the feasibility of long-context inference.

The challenge of MLA adoption is real: it cannot be directly applied to existing GQA models without uptraining, and its inference kernel is more complex than GQA's. However, work such as TransMLA (Zeng et al., 2025) demonstrates that efficient uptraining pathways exist, reducing the barrier to adoption for organisations with existing pretrained GQA checkpoints.

The five-chapter arc of this report traces a coherent engineering progression. Multi-Head Attention established the quality baseline but imposed a KV cache that became the primary constraint on inference efficiency. Multi-Query Attention demonstrated that KV sharing was possible but sacrificed too much quality. Grouped Query Attention found the practical balance between these extremes and became the industry standard. Multi-Head Latent Attention shifts the question entirely, from 'how many representations to store' to 'how compactly each representation can be encoded'. For the next generation of frontier models operating at context lengths of hundreds of thousands to millions of tokens, MLA's approach to KV compression—rather than KV reduction—represents the most promising direction currently known.

## 5.9 The Case for Migration: From GQA to MLA

The engineering case for adopting MLA in new model development is straightforward: the architecture achieves smaller KV cache, higher throughput, and stronger model quality than GQA, as demonstrated by the DeepSeek-V2 results. The more practically urgent question for the broader industry is whether existing GQA-trained models—representing the dominant installed base of production LLMs—can be converted to MLA without the prohibitive cost of full retraining.

The TransMLA framework (Zeng et al., 2025) provides a direct answer and establishes the most complete empirical record to date for GQA-to-MLA migration. Its approach proceeds in two stages: a training-free structural initialisation followed by optional uptraining on a small token budget.

### *Training-Free Conversion*

The first stage requires no gradient updates. For a GQA model with n_g KV groups, each group's Key and Value projection matrices W^K_g ∈ ℝ^(d_h × d) are factorised using singular value decomposition (SVD) into a down-projection and up-projection pair that approximates the original weights: W^K_g ≈ W^UK_g × W^DKV. The resulting matrices initialise the corresponding MLA parameters. Because SVD finds the optimal low-rank approximation of a matrix, this initialisation preserves the pre-trained model's behaviour as closely as the chosen compression rank allows.

Zeng et al. (2025) demonstrate that this training-free conversion of LLaMA-2-7B compresses the KV cache by 68.75 percent while incurring only a 1.65 percent performance drop across six standard benchmarks. This is a compelling result in isolation: no additional training, no GPU hours beyond a one-time SVD decomposition, and most of the KV memory savings of MLA are already realised.

### *Uptraining for Full Recovery*

The second stage is uptraining: continued gradient-based training of the initialised MLA model on a standard language modelling corpus. This allows the model to adapt its weights to the new attention structure and recover any quality degraded by the lossy SVD approximation. The critical engineering finding of TransMLA is how little data this requires.

At a KV compression rate of 93 percent—equivalent to reducing the KV cache to MLA's full compression depth—uptraining LLaMA-2-7B on approximately 6 billion tokens recovers performance to within the margin of the original GQA model across standard benchmarks, with a maximum degradation of 0.5 percent on LongBench. For reference, LLaMA-2-7B was pretrained on two trillion tokens. Six billion tokens therefore represents 0.3 percent of the original pretraining budget: roughly one-third of one percent. At the same 93 percent compression rate, the resulting model achieves a 10.6× inference speedup at 8,192-token context length relative to the original GQA model.

This figure—approximately 0.3 to 0.6 percent of the original pretraining data, depending on the base model's total token budget—is the empirical basis for the claim that GQA models can be converted to MLA at a small fraction of the original training cost. The user intuition that 'five percent' represents the rough scale of uptraining required is broadly consistent with this finding: at larger model scales with longer pretraining runs, the 6B token budget represents a somewhat larger fraction of the total, and the precise percentage varies with model size and target compression ratio.

A concurrent approach, MHA2MLA, achieves similar KV cache compression but incurs a 21.85 percent performance decline at equivalent compression rates—substantially worse than TransMLA's 0.5 percent drop. The architectural difference lies in TransMLA's SVD-based initialisation, which provides the uptraining phase with a significantly better starting point by preserving the pretrained weight structure rather than reinitialising attention projections from random weights.

### *Industry Implications*

The practical significance of these results extends beyond the specific models tested. The GQA-to-MLA migration pathway implies that the substantial compute investment already made in training GQA models—LLaMA, Mistral, Qwen, Gemma, and their derivatives—need not be abandoned to benefit from MLA's inference efficiency. An organisation with an existing pretrained GQA checkpoint can run the TransMLA SVD initialisation (a one-time offline operation requiring no GPU training), immediately deploy a model with reduced KV cache at the cost of a 1.65 percent quality penalty, and optionally close most of that gap with uptraining on 6B tokens.

The converted model is structurally identical to a natively trained MLA model and is fully compatible with DeepSeek's inference infrastructure, including the SGLang and vLLM inference backends that have been optimised for the MLA attention kernel. This compatibility means that converted models can immediately leverage hardware-level optimisations developed for DeepSeek-V2 and subsequent MLA-based models, without requiring separate kernel development.

Taken together, the DeepSeek-V2 architecture results and the TransMLA uptraining findings make a coherent engineering argument: MLA is not merely an architectural curiosity applicable only to models trained from scratch under exceptional compute budgets. It is a practical upgrade path for the existing ecosystem of GQA-trained models, achievable at a cost that is well within the budget of typical fine-tuning workloads. As context windows continue to extend and the KV cache bottleneck becomes more acute, the case for this migration will strengthen.

## References

[1] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention Is All You Need. Advances in Neural Information Processing Systems (NeurIPS), 30. arXiv:1706.03762.

[2] Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. IEEE Transactions on Neural Networks, 5(2), 157–166.

[3] Clark, K., Khandelwal, U., Levy, O., and Manning, C. D. (2019). What Does BERT Look at? An Analysis of BERT's Attention. Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP. arXiv:1906.04341.

[4] Voita, E., Talbot, D., Moiseev, F., Sennrich, R., and Titov, I. (2019). Analyzing Multi-Head Self-Attention: Specialized Heads Do the Heavy Lifting, the Rest Can Be Pruned. Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL). arXiv:1905.09418.

[5] Shazeer, N. (2019). Fast Transformer Decoding: One Write-Head is All You Need. arXiv:1911.02150.

[6] Ainslie, J., Lee-Thorp, J., de Jong, M., Zemlyanskiy, Y., Lebron, F., and Sanghai, S. (2023). GQA: Training Generalised Multi-Query Transformer Models from Multi-Head Checkpoints. Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP). arXiv:2305.13245.

[7] DeepSeek-AI (2024). DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model. arXiv:2405.04434.

[8] Zeng, M., Zhang, H., Mao, H., and Xiong, C. (2025). TransMLA: Multi-Head Latent Attention Is All You Need. Advances in Neural Information Processing Systems (NeurIPS), 2025. arXiv:2502.07864.

[9] Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., et al. (2023). Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288.

[10] Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Devendra, S. C., de las Casas, D., Bressand, F., Lengyel, G., et al. (2023). Mistral 7B. arXiv:2310.06825.

[11] Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Re, C. (2022). FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. Advances in Neural Information Processing Systems (NeurIPS), 35. arXiv:2205.14135.

[12] Dao, T. (2023). FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. International Conference on Learning Representations (ICLR), 2024. arXiv:2307.08691.

[13] DeepSeek-AI (2024). DeepSeek-V3 Technical Report. arXiv:2412.19437.

[14] Su, J., Ahmad, M., Ahmed, S., Wan, S., Liu, Y., and Liu, S. (2021). RoFormer: Enhanced Transformer with Rotary Position Embedding. Neurocomputing, 568. arXiv:2104.09864.

[15] Williams, S., Waterman, A., and Patterson, D. (2009). Roofline: An Insightful Visual Performance Model for Multicore Architectures. Communications of the ACM, 52(4), 65–76.