@April 30, 2026

Inference Engineering (Lecture 3)

Today's topics:

- Multi-Query Attention (**MQA**)
- Group Query Attention (**GQA**)
- Multi Latent Attention (**MLA**)
- Sparse Attention (**DSA**) → introduced in DeepSeek V3.2

Recap

1. Heart of Inference Engineering → KV Cache.
2. The need of KV Cache from first principles.
3. Having multiple Heads captures more perspectives, grammar, subject, verb etc.
4. Context matrix goes through Final Layer Norm and Output Layer and then you have final Logits matrix.
5. To predict the next token, you need to look at the last row of Logits. You apply Softmax and find the token with the maximum probability.
6. With the new token, the cycle repeats but there are huge number of redundant computations. Why do we have to compute these again and again? It hurts us in the order of cubic manner → increases wall clock time exponentially.
7. We realize that you need all the past keys, but you need only one query vector. You do not need the previous queries at all. So you need the Keys matrix and Values matrix and major part of these matrices can be cached.
8. The loop repeats until `<EOS>` token is reached, `max_new_tokens` is reached, session terminated or crashed. Or it may happen that in vLLM the engine itself has defined maximum number of new tokens based on how many things can fit in KV cache and if it exceeds that for each user we have to stop.
9. Benefits of KV cache is it speeds up the inference for us. If we do not cache, the number of redundant operations will be in cubic order, that is, $O(n^3)$ which in turn increases wall clock time exponentially.

10. The reason you need to cache the keys is because when new token comes, you definitely need to find the relation of a token with its past. So, you need all the keys.
11. The reason you need to cache the values is whenever you want to find the context vector, it is the weighted summation of all the previous tokens.
12. Evil of KV cache is the memory which it takes in VRAM or the HBM. It has to be stored in HBM and that is the real problem. They have to be brought to place where compute actually happens. And this affects GPU roofline plot.
13. Formula for KV cache size: $l \times b \times n \times h \times s \times 2 \times 2 \text{ bytes}$, for $FP16$

Multi-Query Attention

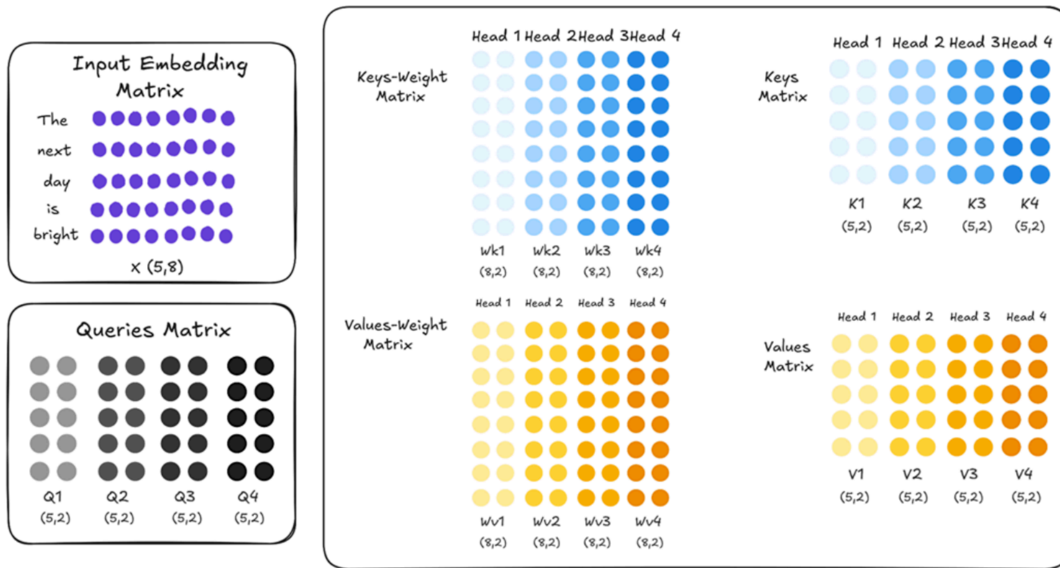
In this lecture, we will first try to reduce the KV memory by attacking the number of attention heads, n . We are on this mission to reduce the KV Cache. Let's start this journey. There are many ways to reduce KV cache, but let's start with what people did first.

Traditional MHA

Let's say you have input "*The next day is bright*" and we get **Input Embedding Matrix** $X(5,8)$ which is now going into the **Transformer** block. It multiplies with **trainable Keys matrix** $(W_k(8,8))$ to obtain **Keys** matrix $(K_{(8,8)})$ and also multiplies with **trainable Value matrix** $(W_v(8,8))$ to obtain **Values** matrix $(V_{(8,8)})$.

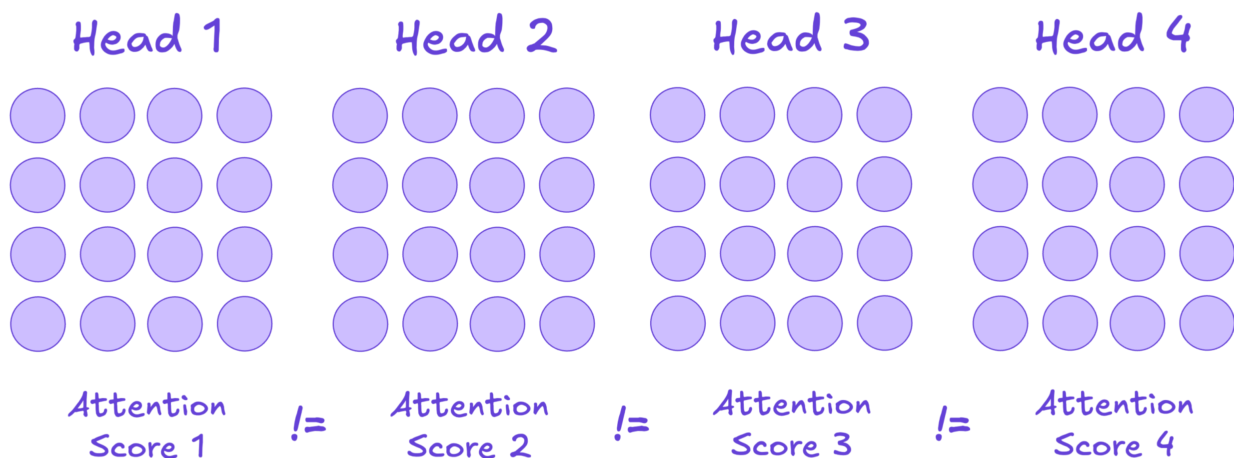
In this example, these **trainable Keys matrix**, **trainable Values matrix**, **Keys** and **Values** have 4 **Heads** each with $(8,2)$ dimensions. Meaning **Head 1** → W_{k1}, W_{v1}, **Head 2** → W_{k2}, W_{v2}, **Head 3** → W_{k3}, W_{v3} , **Head 4** → W_{k4}, W_{v4}. Consequently, for the **Keys** and **Values** as well.

In traditional Multi-Head Attention the trainable weights, $W_{k1} \neq W_{k2} \neq W_{k3} \neq W_{k4} \quad (8,2)$ and $W_{v1} \neq W_{v2} \neq W_{v3} \neq W_{v4} \quad (8,2)$ and hence $K_1 \neq K_2 \neq K_3 \neq K_4 \quad (5,2)$ and $V_1 \neq V_2 \neq V_3 \neq V_4 \quad (5,2)$. Meaning $\text{Head 1} \neq \text{Head 2} \neq \text{Head 3} \neq \text{Head 4}$



Why do we want the weights to be different? To capture and learn all the perspectives.

Since you have 4 **Attention Heads**, you will have 4 different **Attention Score** matrices and each of them will be $(4, 4)$ matrix. These are different from each other. So, each **Head** essentially captures a different nuance or perspective through the **Attention Scores**.



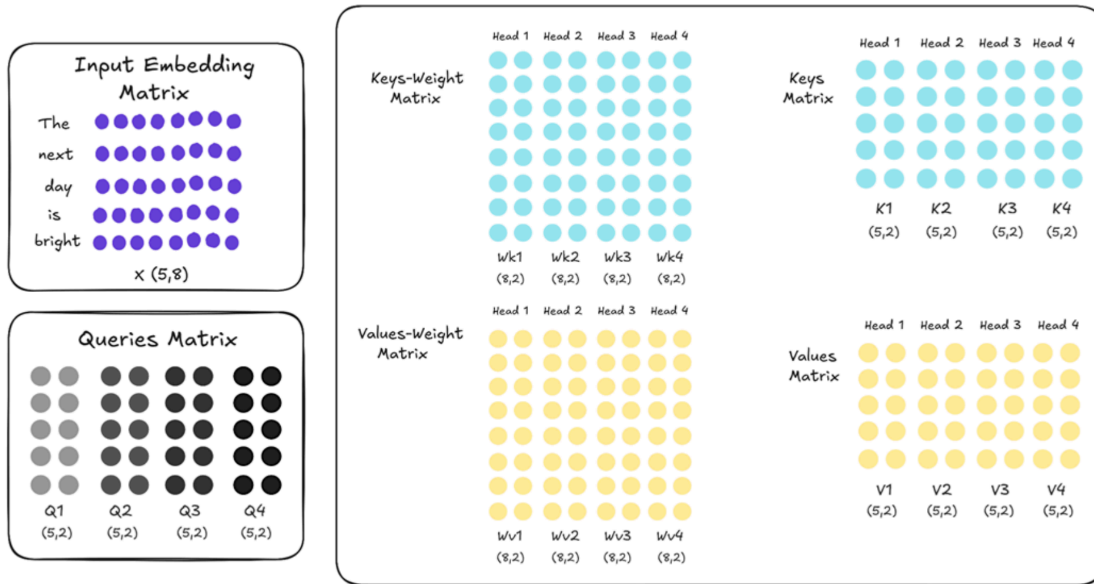
The Lazy Part - twist from MHA

Since the weights for **Keys** and **Values** are different in different **Heads**, we need to cache all of these since they are different. We have to cache the **Keys** matrix for all the **Heads** and we need to cache the **Values** matrix for all the **Heads**. That is why there is the n term in KV Cache formula.

What people figured out is that, what if all the **Heads** have the same values?

Meaning $W_{k1} = W_{k2} = W_{k3} = W_{k4} \quad (8, 2)$ and $W_{v1} = W_{v2} = W_{v3} = W_{v4} \quad (8, 2)$

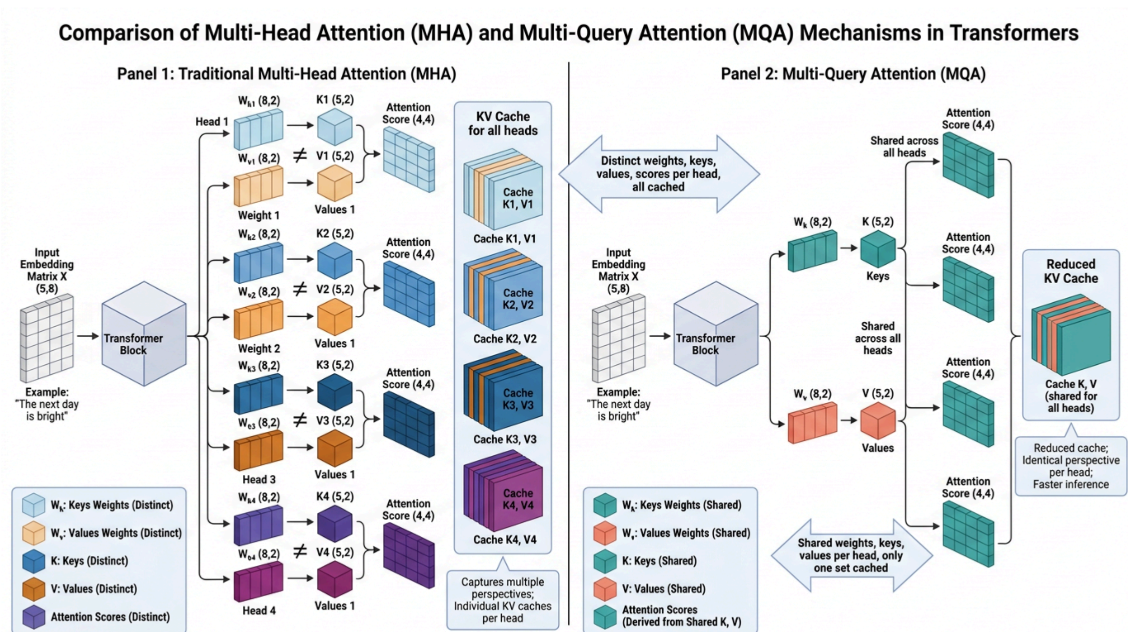
Thus, $K_1 = K_2 = K_3 = K_4 \quad (5,2)$ and $V_1 = V_2 = V_3 = V_4 \quad (5,2)$



With this, you only need to cache K_1, V_1. You do not need to cache $K_2, V_2, K_3, V_3, K_4, V_4$

So instead of caching all of the 4 **Keys** and **Values**, you just cache a fraction of it. So here itself you get the reduction by the number of **Heads**.

This is **Multi-Query Attention**.



▼ In MQA, are the Key and Value projection weights shared across all heads? If so, does that mean all heads end up using the same Keys and Values, and therefore produce identical attention scores and identical context outputs? If not, where does the diversity between heads come from?

Expanding the question

In standard multi-head attention (MHA), each attention head has its own trainable projection matrices for Queries, Keys, and Values. Because each head learns different weights, it produces different K and V representations, leading to different attention patterns and different contextual views of the input.

In multi-query attention (MQA), are the Key and Value projection weights shared across all heads? If so, does that mean all heads end up using the same Keys and Values, and therefore produce identical attention scores and identical context outputs? If not, where does the diversity between heads come from?

Analogy

Imagine a classroom with **8 students (heads)** reading the same book.

In Multi-Head Attention (MHA)

Each student gets:

- their own **question style** (Q)
- their own **highlighter style** (K)
- their own **note-taking style** (V)

So:

Student 1: *"Find emotional parts"*

Student 2: *"Find action scenes"*

Student 3: *"Find names and places"*

Since everyone highlights differently and takes notes differently:

- Keys differ
- Values differ
- Attention patterns differ
- Final context differs

Everyone sees the story from a different angle.

In Multi-Query Attention (MQA)

Now the school wants to save memory.

So they say:

"All students must share one common highlighted book and one common notebook."

Meaning:

- Queries are still different per student/head ✅
- Keys are shared ❌ (same K)
- Values are shared ❌ (same V)

So all students look at the **same highlighted text and same notes**, but each asks **different questions**.

Student 1 asks:

> "Where are emotional moments?"

Student 2 asks:

> "Where are action scenes?"

Student 3 asks:

> "Where are names?"

Even with identical K and V:

The attention score is: QK^T

The Q **is still different for each head**.

So:

Head 1: Q_1K^T

Head 2: Q_2K^T

Head 3: Q_3K^T

Different Q → different attention scores. So attention patterns are still different.

But there *is* some loss.

Because values are shared:

After attention weights are computed: $softmax(Q_iK^T)V$

all heads pull information from the same value bank.

So diversity reduces somewhat.

Not:

> identical outputs

but more like:

> different students asking different questions while sharing the same textbook and notebook.

Tiny summary:

Component	MHA	MQA
Q weights	Different per head	Different per head
K weights	Different	Shared
V weights	Different	Shared
Attention scores	Different	Still different
Context outputs	Different	Still different, but less diverse
KV cache memory	Large	Much smaller

That KV sharing is why MQA became popular in large inference systems: huge reduction in KV cache size while keeping most of the quality.

▼ **Doesn't this impact hugely on accuracy? Lets say input "*The next day is bright*", then assume $d = 8$, so we have $(5, 8)$ as input, it multiplies with W_k and W_v having $(8, 8)$ dimension and we get K and V with. $(8, 8)$. This is multi head considering there are 4 Heads each head with $(8, 2)$ but all heads are different so it is okay. Now with MQA, all $W_{k1} = W_{k2} = W_{k3} = W_{k4}$ meaning only $(8, 2)$ needs to be cached for K and so for V isn't it? Meaning, we got reduction by factor of 4 in heads right? Then in this case accuracy should have gone for toss. From $(8, 8)$ cache to $(8, 2)$. Is my understanding wrong?**

So the intuition with question is:

> "We compressed K/V by $4\times$. Shouldn't quality collapse?"

Let me first correct one shape detail, then explain why quality does **not** fall apart.

For input:

> "The next day is bright"

Suppose sequence $length = 5$ and model dimension: $X : (5, 8)$

4 heads, so: $d_{head} = 2$

In standard MHA:

Queries: $W_Q : (8, 8)$

Keys: $W_K : (8, 8)$

Values: $W_V : (8, 8)$

After projection: $Q, K, V : (5, 8)$

Then split:

Head1: $(5, 2)$

Head2: $(5, 2)$

Head3: $(5, 2)$

Head4: $(5, 2)$

Each head has its own K/V slice.

KV cache stores: $4 \times (5, 2)$ for K and similarly for V.

Total: $(5, 8)$

In **MQA**:

Queries still remain per head: $Q : (5, 8)$

split into: $4 \times (5, 2)$

But K and V become: $K : (5, 2)$, $V : (5, 2)$ shared by all heads.

So yes:

instead of caching: $4 \times (5, 2)$

you cache only: $1 \times (5, 2)$

Reduction: $4\times$

Exactly what you said.

Now your question:

> “Wouldn't accuracy go for a toss?”

Surprisingly: not much.

Here's why.

The huge diversity of attention mostly comes from Q, not K/V.

Each head still computes:

Head1: $Q_1 K^T$

Head2: $Q_2 K^T$

Head3: $Q_3 K^T$

Head4: $Q_4 K^T$

Even though K is shared.

Different Qs ask different questions.

Think:

Shared K says:

> token1=The
>
> token2=next
>
> token3=day
>
> token4=is

> token5=bright

Head 1 asks:

> "Which token describes time?"

Head 2 asks:

> "Which token is adjective-related?"

Head 3 asks:

> "Which token modifies another?"

Different questions on the same memory.

So attention maps still differ.

The deeper reason is something researchers discovered:

Many K/V heads in large models were redundant.

People inspected trained transformers and found:

- several heads attend similarly
- many K/V patterns overlap
- not every head learns a unique representation

So MQA says:

> “If K/V heads are repeating information anyway, why keep many copies?”

It turns out much of that capacity was unused.

There *is* some quality loss.

Approximate trend:

MHA: 100%

MQA: $97–99\%$

But KV memory: 100% → 25% for 4 heads.

For very large models: memory savings are enormous.

Inference throughput can become dramatically better.

That's why modern models often use a middle ground called **Grouped Query Attention (GQA)**.

Instead of:

$4\ Q$ heads → $1\ KV$ head

they do:

$4\ Q$ heads → $2\ KV$ heads

or:

32 Q heads → 8 KV heads

This preserves more diversity.

Many modern LLMs such as Meta Platforms's models and others moved toward GQA because it keeps almost all quality while reducing cache cost.

Your mental model of "**$4\times$ cache reduction**" is correct. The only thing to adjust is:

You are compressing only K/V, not all representational power. Q remains head-specific, and that turns out to preserve much more intelligence than people first expected.

Trade Offs

You're essentially asking:

> "If MQA gives ~4× or more KV cache reduction and little quality loss, why doesn't everyone always use it?"

The answer: **MQA optimizes memory and throughput, but sacrifices representational richness.**

What MQA improves

For a model with:

- 32 attention heads
- sequence length = 32k
- long conversations

KV cache dominates memory.

MHA cache: $32 \times K + 32 \times V$

MQA: $1 \times K + 1 \times V$

Massive reduction.

This gives:

- lower **VRAM** usage
- more concurrent users
- faster decoding throughput
- longer contexts possible
- lower cost

This is huge in production.

Actual Trade-Offs

1. Less head specialization

In **MHA**:

Head 1 may track:

> subject ↔ verb relation

Head 2:

> chronology

Head 3:

> code syntax

Head 4:

> entity references

Each has its own K/V memory.

With **MQA**:

All heads share one K/V representation.

Different Qs ask different questions, but everyone reads from the same memory bank. So information becomes more compressed. Some nuanced distinctions can blur.

Example:

Sentence:

> "The trophy doesn't fit into the suitcase because it is too large."

Some heads may need:

- object reasoning
- pronoun resolution
- spatial reasoning

Separate K/V memories help specialization. Shared K/V can weaken this. Not catastrophic, but measurable.

---

2. Longer contexts expose weaknesses

For short contexts:

MQA works very well.

For:

- 64k context
- 128k context

- large codebases
- multi-document reasoning

tiny errors accumulate.

Because K/V cache is the actual memory of previous tokens.

Compressing memory too aggressively may lose subtle distinctions.

---

3. Retrieval becomes less expressive

Think of K as an indexing system.

MHA:

32 librarians create 32 indexes.

MQA:

1 librarian creates 1 index.

Queries differ:

> "Find science books"

> "Find history books"

but everyone searches the same index.

Sometimes specialized indexes help.

---

4. Quality drops more at smaller model sizes

Large models have lots of redundancy.

Smaller models:

- 1B
- 3B
- 7B

have less spare capacity. Removing K/V diversity hurts more. Large models often tolerate **MQA** better.

Why GQA became popular

People realized:

MHA → too expensive

MQA → slightly too compressed

Middle ground:

32 Q heads

8 KV heads

instead of:

$32\ Q$ heads

$1\ KV$ head

You get:

- ~$4\times$ cache savings
- almost **MHA** quality
- better specialization

That is why many modern systems moved to **Grouped Query Attention**.

Inference engineers basically discovered:

> "Full **MHA** is wasteful; full **MQA** can be too aggressive; **GQA** lands in the sweet spot."

That is why inference architecture today is often more about **memory economics** than raw modeling power.

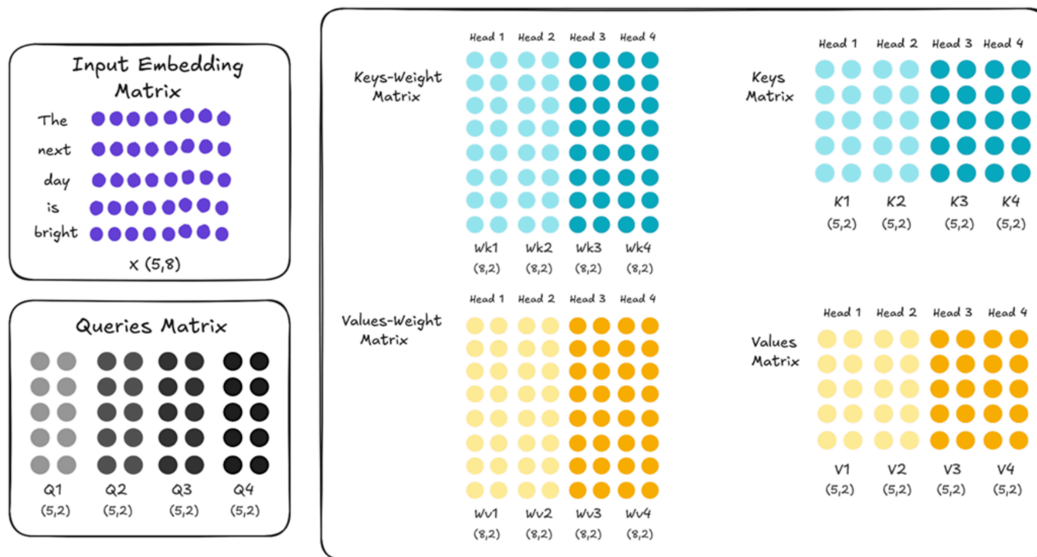
Grouped Query Attention

There has to be something in the middle between MHA and MQA. If you were to come up with something what would you do mathematically? We need to have more heads also and also save KV cache.

There is no LLM which is reliably good which uses MQA. What we will learn right now is actually used by many models including Llama which is called **Grouped Query Attention**.

It is the best of both worlds.

Instead of copying these values across all the heads, we'll form groups. For example, **Head 1** and **Head 2** becomes **Group 1** and **Head 3** and **Head 4** becomes **Group 2**. Within one group all the **Heads** will have the same content.



Here identical contrasted color heads are grouped into one in both trainable weights, keys and values

We can see that the KV cache size is reduced by the factor of 2 in the above example as we have 2 groups. And we just have to store a representative value for each group. Meaning, for **Group 1**, we just have to store **Head 1**, and for **Group 2**, we just have to store **Head 3**. Instead of store for all the heads, we just store for each group now.

The KV cache formula now for different attention becomes:

$$\text{MHA} = L \times B \times N \times S \times H \times 2 \times 2$$

$$\text{MQA} = L \times B \times 1 \times S \times H \times 2 \times 2 \quad \text{reduction factor of } 1/N$$

$$\text{GQA} = L \times B \times G \times S \times H \times 2 \times 2 \quad \text{reduction factor of } \ G/N$$

Although **GQA** is having less reduction faction it is a good compromise as we are not sharing everything across **Heads** now. Each Group can still retain a different perspective.

Researchers perform ablation study for the number of Group sizes and then find the accuracy and make plot of what Group size maintains the accuracy but still reduces the size of KV cache.

Multi-Head Latent Attention (MLA)

Implemented by DeepSeek team in January 2025 and is one of the biggest innovations. Why?

Because on one hand, we have **MHA** → has great performance, but massive memory cost.

On other hand, we have **MQA** → has great memory savings, but poor performance

DeepSeek wanted to achieve the best of both worlds. They asked themselves, "*Can we get a low cache size and can we have high performance?*". They solved this problem by introducing the latent attention concept.

DeepSeek team did not group across **Keys** and **Values**. They shifted the focus from reducing the number of **Heads** to compressing the information within these **Heads**. Instead of caching K and V

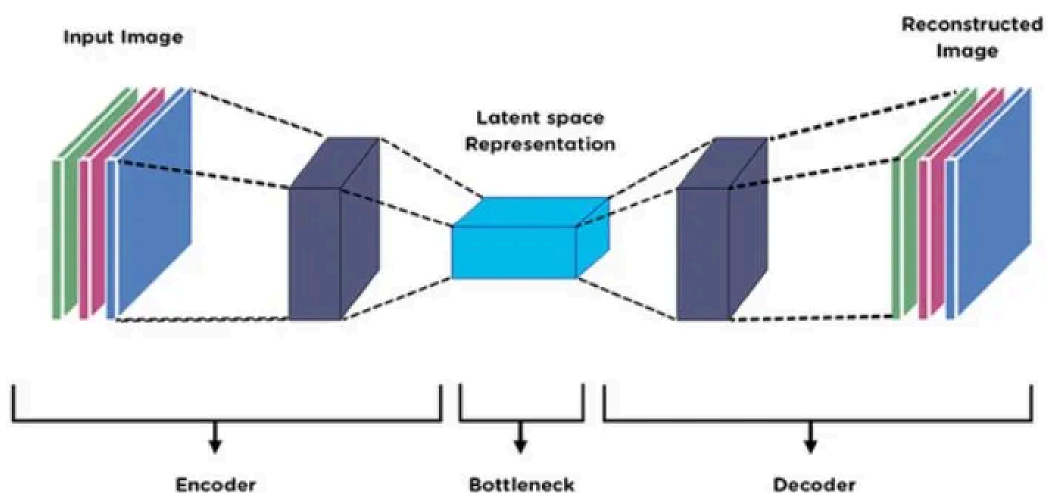
separately, they thought what if we could first project input into a single, combined, much smaller matrix, called a **Latent Matrix** and cache only that?

Instead of caching two large matrices, that is, K and V, we only cache one smaller, lower dimensional matrix called C_{kv}. This single matrix becomes our highly efficient cache.

When we need the full K and V, we can reconstruct them on the fly from compressed latent representation.

This idea is not new, it was implemented in encoding and decoding in vision or PCA or Variational Auto-encoders etc, projection is very commonly used.

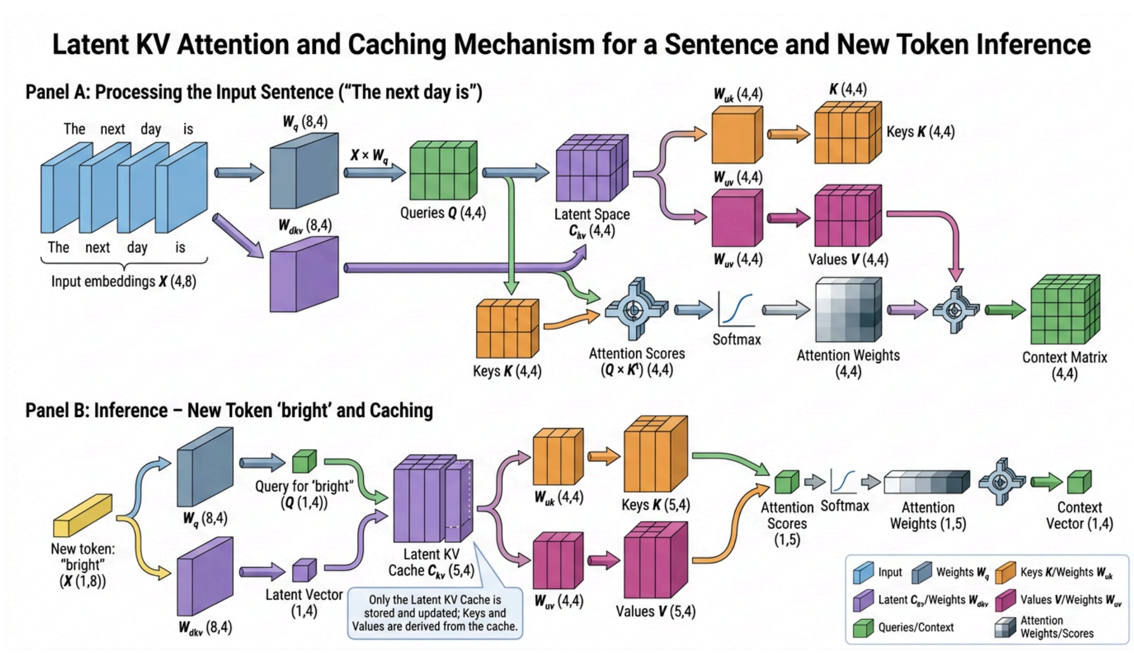
▼ **A Perfect guide to Understand Encoder Decoders in Depth with Visuals**



To read more click here

The sentence is "*The next day is*" and this is input matrix which is $(4, 8)$ and let's call it X. The first thing we do is that we multiply with $W_q(8, 4)$ which is trainable weight matrix to get Query matrix $Q(4, 4)$. Until here nothing has changed. Now, we will multiply $X(4, 8)$ with $W_{dkv}(8, 4)$ which is **Down Projection Matrix** which will take us to **Latent Space** $(4, 4)$. When we multiply we get a **Latent Space** matrix called $C_{kv}(4, 4)$. To get Key and Value vectors, we take the $C_{kv}(4, 4)$ and multiply with the **Up Projection Keys Matrix** $W_{uk}(4, 4)$ and multiply with the **Up Projection Values Matrix** $W_{uv}(4, 4)$ to get the **Keys** $K(4, 4)$ and the **Values** $V(4, 4)$. From here, nothing changes, so the Queries $Q(4, 4)$ multiplied with the **Keys** transpose $K^T(4, 4)$ and get the **Attention Scores** $(4, 4)$. And then we get **Attention Weights** $(4, 4)$. Then the Values matrix $V(4, 4)$ is multiplied with the **Attention Weights** $(4, 4)$ to get **Context Matrix** $(4, 4)$. Now let's see what and how inference happens when new token comes in. Let's say a new token comes in which is "*bright*", and the vectorial representation for this new token will be $(1, 8)$. Now this is $X(1, 8)$ and we will multiply with $W_q(8, 4)$ to get **Query Vector** for token "*bright*" which is of course $Q(1, 4)$. Now $X(1, 8)$ is also multiplied with $W_{dkv}(8, 4)$ which will give a **Latent Vector** $(1, 4)$ for the token "*bright*". Now we are going to cache this **Latent Vector** $L(1, 4)$ by appending to the previously cached **Latent Vector** $(4, 4)$ and have new

updated **Latent KV Cache** matrix $(5,4)$. We will not cache Key or Value matrices separately. Now, we will multiply again with $W_{uk}(4,4)$ and multiply with $W_{uv}(4,4)$ which will give us **Keys** $(5,4)$ and **Values** $(5,4)$ respectively. After this we know what to do, that is, we have **Query** vector $Q(1,4)$ multiplied with **Keys** transpose $K^T(4,4)$ and get the **Attention Scores** $(1,4)$ for "*bright*", then **Scaling**, **Softmax** and get **Attention Weights** $(4,4)$ which is again multiplied with **Values** matrix $V(4,4)$ and get **Context Vector** $(1,4)$ for "*bright*". The only thing we need to cache is **Latent KV Cache**.



KV Cache Reduction

If DeepSeek's embedding matrix is 7168 dimensions, if there is no MLA, for both K and V we had to cache 7168 dimensions, that would be, $2 \times d = 2 \times 7168$

With MLA, let's say the dimension is only 512, we just have to cache 1 matrix with a dimension of 512. So we get the reduction factor of 2, because instead of 2 matrix we cache only 1 matrix. And we also get a reduction factor of $7168/512$

$\text{Total Reduction} = 2 \times \dfrac{7168}{512}$ which is a huge reduction of KV cache.

So we are not destroying information across different heads now. We are actually retaining the information across different heads. We did not even reduce the number of **Heads**.

For $\text{MHA} = L \times B \times N \times H \times S \times 2 \times 2 \quad N \times H \rightarrow \text{embedding dimension}$

For $\text{MLA} = L \times B \times D_{latent} \times S \times 1 \times 2$

$\text{Reduction} = \dfrac{N \times H}{D_{latent}} \times 2$

DeepSeek's actual model has embedding dimension 16384, then:

$N \times H = 16384$

$D_{latent} = 512$

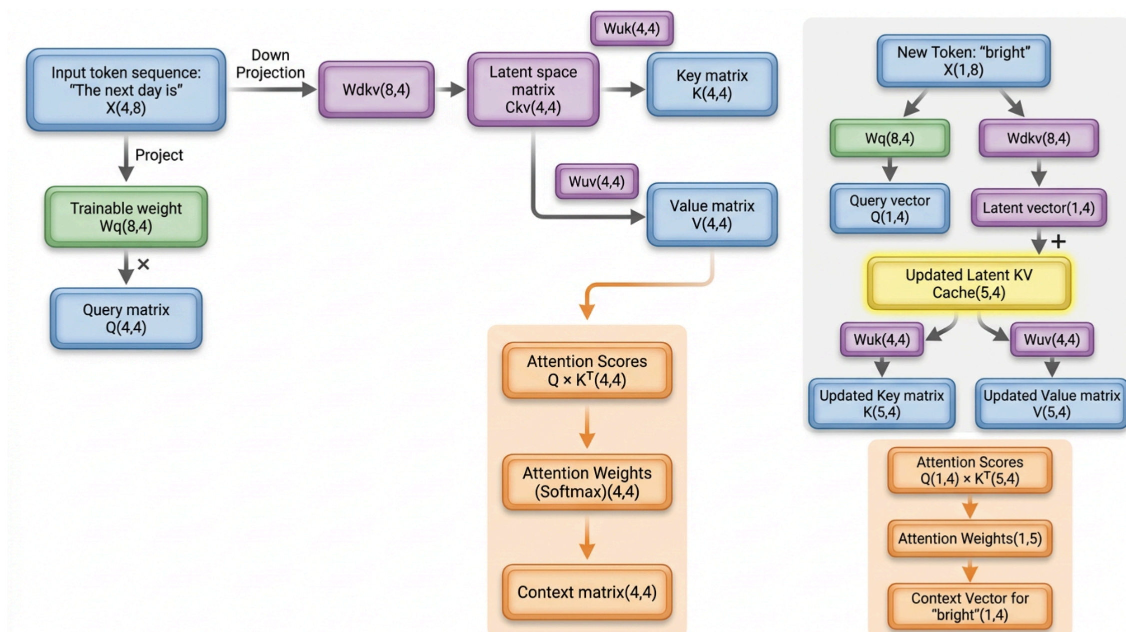
$\text{Reduction} = \frac{16384}{512} \times 2 = 64 \text{ times}$

This is incredible achievement because, if you did not use Latent Attention, then the KV cache size would have been around $400GB$. With **MLA**, the same KV cache now becomes only $7GB$ and performance is also not hampered.

Preserving Performance: Why head diversity maintained?

Up projection matrices: W_{uk} and W_{uv}
These up project matrices are Multi-Head! This means that for each head, the content in these matrices are different. And this is fundamentally different from GQA and MQA.
When the Attention calculation is made, every Head is working with it's unique K and V matrices, just like MHA.
We have not shared anything across heads. We have preserved the full diversity and expressive power of the Multi-Head design.
This is how MLA archives the best of both worlds.

Architecture Flow Diagram for MLA



▼ **Are the number of trainable parameters the same?**

The number of trainable parameters actually increase slightly because you have now additional trainable weight matrices. But that is completely fine because anyways during inference you are in the decode phase (memory bound). Even if FLOPs increase slightly, it is totally fine because anyways there are so many FLOPs left of the GPU to utilize. The key thing is to reduce the number of bytes transferred from the HBM to the cores where the computation actually happens.

▼ **What is the difference between weights and activations?**

Weights = learned parameters of the model

They:

- are stored permanently
- get updated during training
- define the model's behavior

Examples:

- W_q
- W_k
- W_v
- W_o
- W_{dkv}
- W_{uk}
- W_{uv}

These are trainable matrices.

Activations = temporary computed values during a forward pass

They:

- depend on the current input token
- are generated dynamically
- are NOT stored as permanent knowledge

Examples:

- Queries (Q)
- Keys (K)
- Values (V)
- hidden states
- MLP outputs

These are activations.

Simple intuition

Think of a neural network like a factory.

Weights = machinery configuration → Permanent.

Activations = products moving through conveyor belts → Temporary.

In attention specifically

Suppose token embedding is: x, then transformer computes:

$Q = xW_q$

$K = xW_k$

$V = xW_v$

Important distinction

Here:

- W_q, W_k, W_v → weights (trainable parameters)
- Q, K, V → activations (computed outputs)

Why are K and V activations?

Because they depend on the input sentence.

Example:

Input 1:

```
"The cat sat"
```

produces one set of K/V.

Input 2:

```
"Quantum physics"
```

produces completely different K/V.

So K/V are dynamic runtime values.

In KV cache

What gets cached during inference?

NOT the weights.

Weights stay fixed in GPU memory already.

What gets cached:

- K activations
- V activations

for previous tokens.

That’s why it’s called KV cache.

Now relating this to MLA (Multi-head Latent Attention)

In MLA, papers often introduce matrices like:

- W_{dkv}
- W_{uk}
- W_{uv}

These are still trainable weights.

They are learned projection matrices.

Example MLA flow

Instead of directly storing huge K/V:

MLA compresses them into latent representation.

Example idea:

Original hidden state: h

Compress: $C_{kv} = h \cdot W_{dkv}$

Here:

- W_{dkv} = trainable weight
- C_{kv} = activation (latent compressed KV)

Then reconstruct keys:

$K = C_{kv} \cdot W_{uk}$

Reconstruct values:

$V = C_{kv} \cdot W_{uv}$

Again:

- W_{uk}, W_{uv} are weights

- (K, V) are activations

Extremely important mental model

Weights answer:

> "What did the model learn?"

Activations answer:

> "What is the model thinking right now for this input?"

Another intuition

Weights → Long-term memory

Activations → Working memory

During training

Training modifies:

- weights

using gradients.

Activations are recomputed every forward pass.

During inference

Weights:

- frozen

Activations:

- continuously generated token-by-token

Why MLA focuses on activations

KV cache problem is about:

- runtime memory
- activation storage

NOT weight storage.

The huge memory cost during long-context inference comes from:

- storing K/V activations for every token.

That's why MLA compresses activations.

Final mapping

Symbol	Type
W_q	Weight
W_k	Weight
W_v	Weight
W_{dkv}	Weight
W_{uk}	Weight
W_{uv}	Weight
Q	Activation
K	Activation
V	Activation
latent KV	Activation
hidden states	Activation

▼ **How Weights, Activations and KV cache move through GPU memory hierarchy?**

This is the heart of modern LLM inference systems. The question refers to the distinction between:

- **static model parameters (weights)**

  vs

- **dynamic runtime state (activations/KV cache)**

and how they move through GPU memory hierarchy.

During inference, what exists?

There are mainly 3 things:

Component	Changes per token?	Stored where?
Weights	No	GPU HBM (mostly static)
Activations	Yes	SRAM/registers/HBM temporarily
KV cache	Yes	GPU HBM (persistent across tokens)

---

1. Weights

These are:

- (W_q, W_k, W_v)

- MLP matrices
- embeddings
- output layers

They are loaded ONCE onto GPU memory before inference starts.

Example:

- 70B model
- FP16

Weights may occupy:

- 140 GB

distributed across GPUs.

Important

Weights are usually:

- NOT repeatedly transferred CPU ↔ GPU every token
- already resident in GPU HBM

That's critical for performance.

During each token step

GPU streams weights from:

- HBM → tensor cores / SRAM

for matrix multiplication.

So weights DO move internally inside GPU memory hierarchy.

But:

- not across network
- not CPU-to-GPU repeatedly

unless offloading is used.

---

2. Activations

Activations are temporary runtime computations.

Example:

- hidden states
- Q/K/V for current token

- MLP outputs

These are generated during forward pass.

Lifecycle of activations

Example:

1. Compute activation
2. Use immediately
3. Discard

Most activations live only milliseconds during inference.

Exception: KV cache

K and V activations are special.

Why?

Autoregressive decoding needs past context.

So past K/V cannot be discarded.

They are stored persistently.

KV cache

For every generated token:

Store:

- K activation
- V activation

for every layer.

These accumulate over sequence length.

Example

Suppose:

- 32 layers
- seq len = 32K
- batch = 8

KV cache becomes enormous.

Sometimes larger than activations themselves.

During decoding

For new token:

You compute:

- new Q
- new K/V

Then:

- append K/V to cache
- Q attends against ALL cached K/V

What actually moves during inference?

This is the key.

In decode phase (token-by-token)

The bottleneck is often:

Reading KV cache from HBM

NOT weight loading.

Why?

Because:

- each new token must read huge historical KV tensors

This becomes memory-bandwidth bound.

Simplified flow

For one layer:

Weights

Static:

```
HBM -> SRAM -> Tensor cores
```

KV cache

Dynamic:

```
HBM -> compute units every token
```

And KV size scales with:

- context length
- batch size

Why GQA/MQA helps

Because KV cache traffic dominates decoding.

Reducing KV heads reduces:

- HBM reads
- memory bandwidth pressure

That improves:

- throughput
- latency

Very important distinction

Weights

Scale with:

- model size

KV cache

Scale with:

- sequence length
- batch size

Long-context inference often becomes KV-cache dominated.

Where are things physically stored?

GPU hierarchy

From fastest to slowest:

Memory	Purpose
Registers	Tiny immediate computations

Memory	Purpose
SRAM/shared memory	Tensor tile computation
HBM	Weights + KV cache
CPU RAM	Optional offload
Disk	Model checkpoint storage

Typical inference server

At startup:

1. Load model checkpoint from disk
2. Move weights to GPU HBM

Then during inference:

- activations generated dynamically
- KV cache grows dynamically

Important subtlety

During training:

- activations are also stored for backward pass

During inference:

- most activations discarded immediately
- only KV cache persists

That's why inference memory behavior differs hugely from training.

In distributed inference

Things become more interesting.

Sometimes:

- weights sharded across GPUs
- KV cache sharded separately
- activations communicated between GPUs

This is where:

- tensor parallelism
- pipeline parallelism
- disaggregated inference

- TurboQuant
- paged attention

become important.

Your mental model should now be

Weights = permanent learned brain

Mostly static in GPU memory.

Activations = temporary thoughts

Generated and discarded continuously.

KV cache = remembered conversation history

Dynamic activations that persist across decoding steps.

Live Demo Code Execution

`Clean_KV_Cache_Inference_Benchmark_MHA_GQA_MQA_MLA.ipynb`

Note!

If you understand all these attention mechanisms at the application layer, you can switch the model architecture. If you're using an open-source model, you can decide which architecture you want to go ahead with. But you want to know the trade-offs.

DeepSeek Sparse Attention (DSA)

For **MLA**, what we have seen is we cache only one **Latent Vector** for every token. If there is a sentence "*The next day is*", that is, 4 tokens, then we have one **Latent Vector** for each of these tokens.

That is: $C_{kv} = 4 \text{ tokens} \times 4 \text{ Latent numbers.}$

When you are dealing with Spare Attention, you need to also maintain one more **Cache Vector** per token and that is an **Index Key**. This **Index Key Cache** is maintained to decide which tokens are important and which is not.

That is: $\text{Index K Cache Shape} = 4 \text{ Tokens} \times 4 \text{ Index Numbers}$

Therefore, in **DSA** we store the **Latent Cache** and also the **Index Key Cache**.

Why is this Index used what is the need of this?

The Indexer is just a learned projections. And every **Key** (K) will have an **Index Vector** associated with it.

Essentially what we are going to do is that, whenever a new **Query** (Q) is going to come in, first the Index Query is found for that and then take the dot product of this with all previous keys so that we can choose which of the previous keys are relevant for the current token.

Let's say when a new token "*bright*" comes in. In MLA, what we are doing is that, we first find the **Query Vector** Q associated with this new token (by doing $X(1,8) \times W_q(8,4)$) and we also get **Latent Vector** L (by multiplying $X(1,8) \times W_{dkv}(8,4)$).

But now in **DeepSeek Sparse Attention**, what you do is that when a new **Query Vector** Q comes in, you also multiply it with the **Index Matrix**. so you get a **Lookup Query Vector** $(q_{I_{bright}})$ and **Lookup Key Vector** $(k_{I_{bright}})$for that new **Query Vector** Q.

Why do we need this Look Up vectors? Because what we are now going to do is that, we are going to get the **Look Up Vector** for this **Query**, that is, $(q_{I_{bright}})$ and take dot product of this query with the **Look Up Vector** of all our past **Keys** $(k_{I_{all}})$ and get the **Score** for "*bright*", w.r.t. all other tokens.

$s_i = q_{I_{bright}} \cdot k_{I_i} \quad \forall \text{ i: index keys}$

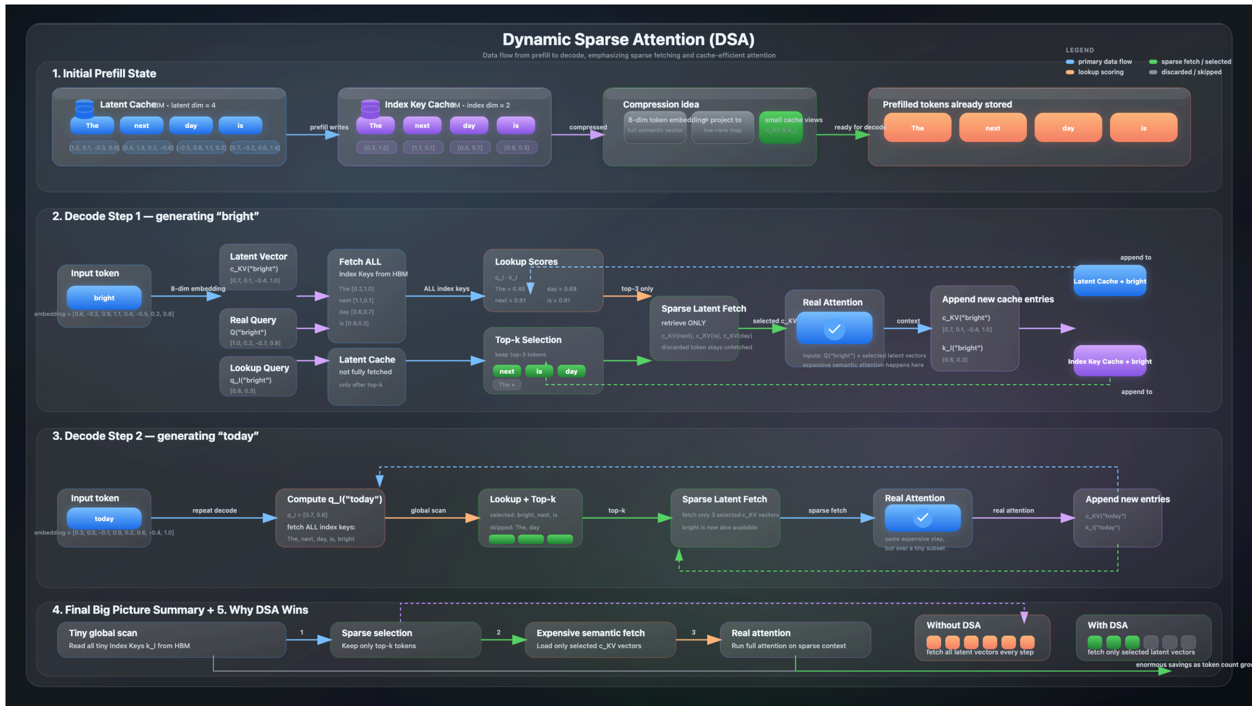
So if s_1 and s_2 are maximum, then we will only choose these tokens. Earlier what happened was every new **Query** which came in that attended to all the past tokens during decode. But now what we're doing is that we get this **Indexing Scores** for all the past tokens and we'll only choose and select those tokens which have these **Maximum** **Indexing Scores**. And we do not consider rest of the tokens at all in our **Attention** mechanism.

Why would we want to do this? Why would we not want to keep all the tokens for every query? If we look back at **MLA**, when we created the **Latent KV Cache** (C_{kv}), all the token's **Latent Vectors** were kept. Now, in **DSA**, all token's **Latent Vectors** will not be kept. Only selective token's **Latent Vectors** will be kept. And they are **Up Projected** to form **Keys** (K) and **Values** (V). And then we get **Attention Scores** which will only be calculated for the selected past tokens for the give **Query** (Q). But the **Attention Scores** for the given **Query** (Q) will now not attend to the tokens which are not important.

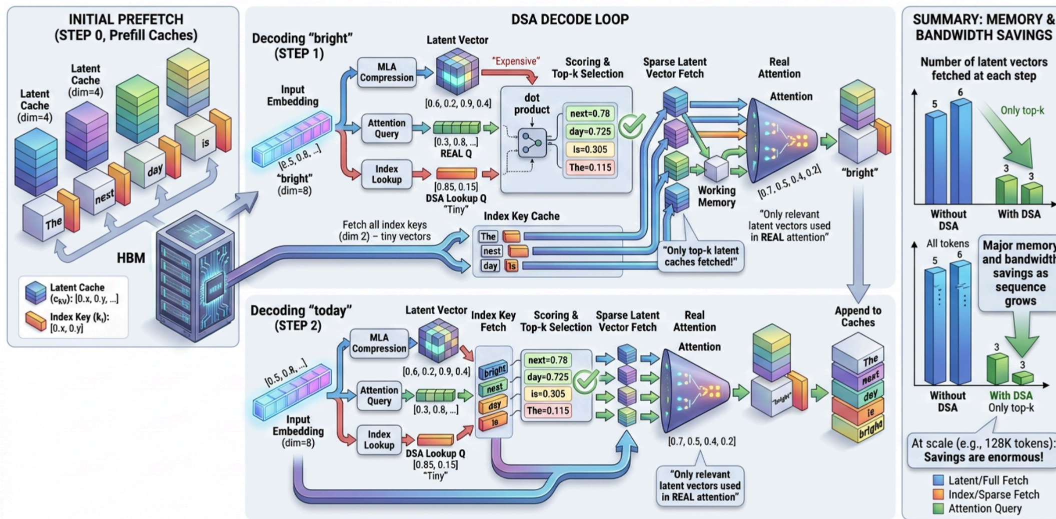
After one token is processed, there are two caches which are updated: **MLA Latent Cache** and **Index Key Cache** is also updated because for every **Key** we need to know its **Indexing Vector** to get the **Score**.

But this is actually increasing our cache compared to MLA right? In **MLA**, we only cache the **Latent Matrix** but here we also need to cache one more called **Index Key Cache**. The KV cache size actually seems to be increasing. **Why is this beneficial to us?** KV cache is increasing but **does all of that KV cache need to be brought to compute form HBM?** **No**. Earlier we naively used to bring all of the KV cache into HBM. But here all of the KV cache does not need to be read from the HBM.

DSA does not beat MLA by shrinking the cache. It beats MLA by reading less during long-context decode.



▼ Complete Walkthrough of DSA with Example



We'll simulate:

- embedding dim (d = 8)
- latent cache dim = 4
- index dim = 2

- top-k = 3

We'll do:

1. Initial prefill
2. Decode token #1
3. Append caches
4. Decode token #2
5. Append caches again

and explicitly track:

- latent cache
- index key cache
- lookup query
- token selection
- which vectors are fetched from HBM

Initial sentence (already prefetched)

```
"The next day is"
```

So initially we already have 4 tokens in cache.

The LAST EXISTING token:

```
"is"
```

is processed through final layer.

Its query attends to previous tokens.

Then output logits are produced.

Example:

Candidate Token	Probability
bright	0.62
rainy	0.11
sunny	0.09

Then decoder selects:

```
"bright"
```

Only AFTER prediction

does `"bright"` become an actual token.

Then:

1. embedding(bright) computed
2. latent vector computed
3. index key computed
4. appended to caches

STEP 0 — Existing caches after prefill

Assume after MLA compression + DSA indexing:

Latent Cache (stored in HBM)

Each token has latent dim = 4

Token	Latent Cache (c_{KV})
The	[0.2, 0.1, 0.5, 0.7]
next	[0.9, 0.3, 0.2, 0.1]
day	[0.8, 0.6, 0.4, 0.2]
is	[0.1, 0.7, 0.9, 0.5]

Index Key Cache (stored in HBM)

Each token has index dim = 2

Token	Index Key (k_I)
The	[0.1, 0.2]
next	[0.9, 0.1]
day	[0.8, 0.3]
is	[0.2, 0.9]

DECODE STEP 1

We now want to generate:

```
"bright"
```

STEP 1A — Input embedding

Suppose embedding for `"bright"` is: x_{bright}

```
[0.5, 0.8, 0.1, 0.3, 0.7, 0.2, 0.4, 0.6]
```

dimension = 8

STEP 1B — Compute latent vector

MLA compression: $c_{KV} = xW_{dkv}$

Suppose result is:

```
c_KV(bright) = [0.6, 0.2, 0.9, 0.4]
```

This is REAL semantic memory.

STEP 1C — Compute REAL query

Normal attention query: $Q = xW_q$

Suppose:

```
Q(bright) = [0.3, 0.8, 0.5, 0.1]
```

Used later for REAL attention.

STEP 1D — Compute lookup query

DSA routing query: $q_I = xW_{qI}$

Suppose:

```
q_I(bright) = [0.85, 0.15]
```

This is tiny and cheap.

STEP 1E — Fetch ALL index keys from HBM

These are tiny vectors.

Fetched:

Token	Index Key
The	[0.1, 0.2]
next	[0.9, 0.1]
day	[0.8, 0.3]

Token	Index Key
is	[0.2, 0.9]

STEP 1F — Compute lookup scores

Dot products:

$$The = 0.085 + 0.03 = 0.115$$

$$next = 0.765 + 0.015 = 0.78$$

$$day = 0.68 + 0.045 = 0.725$$

$$is = 0.17 + 0.135 = 0.305$$

STEP 1G — Top-k selection

top-k = 3

Selected tokens:

Token	Score
next	0.78
day	0.725
is	0.305

Discard:

```
"The"
```

IMPORTANT

Notice:

We ONLY fetched:

- tiny index vectors globally.

We did NOT fetch all latent vectors yet.

STEP 1H — Fetch selected latent vectors ONLY

Now fetch from HBM:

Token	Latent Cache
next	[0.9,0.3,0.2,0.1]

Token	Latent Cache
day	[0.8,0.6,0.4,0.2]
is	[0.1,0.7,0.9,0.5]

We never fetched `"The"` latent vector.

That is the savings.

STEP 1I — REAL attention happens

Now REAL attention uses:

- Q(bright)
- selected latent vectors

instead of all tokens.

Suppose output becomes:

```
Attention output = [0.7, 0.5, 0.4, 0.2]
```

Then model predicts:

```
"bright"
```

STEP 1J — Append caches

Now `"bright"` becomes part of history.

Append to HBM:

Latent Cache

```
[0.6, 0.2, 0.9, 0.4]
```

Index Key Cache

Need index key for future routing.

Suppose:

```
k_I(bright) = [0.95, 0.05]
```

Append this too.

HBM after decode step 1

Latent Cache

Token	Latent
The	[...]
next	[...]
day	[...]
is	[...]
bright	[0.6,0.2,0.9,0.4]

Index Cache

Token	Index Key
The	[...]
next	[...]
day	[...]
is	[...]
bright	[0.95,0.05]

DECODE STEP 2

Now model predicts next token:

```
"today"
```

STEP 2A — Embedding

Suppose:

```
x(today) =
[0.6,0.1,0.7,0.2,0.4,0.9,0.5,0.3]
```

STEP 2B — Compute lookup query

Suppose:

```
q_I(today) = [0.92, 0.08]
```

STEP 2C — Fetch ALL index keys

Now 5 tokens exist.

Fetched:

Token	Index Key
The	[0.1,0.2]
next	[0.9,0.1]
day	[0.8,0.3]
is	[0.2,0.9]
bright	[0.95,0.05]

STEP 2D — Lookup scores

$$bright = 0.92(0.95) + 0.08(0.05) = 0.874$$

Very high.

Suppose final scores:

Token	Score
bright	0.874
next	0.836
day	0.760
is	0.256
The	0.108

STEP 2E — Top-k

Keep:

```
bright
next
day
```

STEP 2F — Fetch ONLY those latent vectors

Fetch from HBM:

Token	Latent
bright	[0.6,0.2,0.9,0.4]

Token	Latent
next	[0.9,0.3,0.2,0.1]
day	[0.8,0.6,0.4,0.2]

Again:

- `"The"` and `"is"` latent vectors never fetched.

STEP 2G — Real attention

Now actual semantic attention happens only on selected tokens.

Model predicts:

```
"today"
```

STEP 2H — Append caches again

Store:

---

latent cache

```
c_KV(today)
```

---

index key cache

```
k_I(today)
```

FINAL BIG PICTURE

At EVERY decode step:

---

1. Tiny global scan

Fetch:

```
ALL index vectors
```

Cheap.

---

2. Sparse selection

Choose top-k relevant tokens.

---

3. Expensive semantic fetch

Fetch:

```
ONLY selected latent vectors
```

This is where savings happen.

---

4. Real attention

Only on selected tokens.

WHY DSA WINS

Without DSA:

At step 2:

```
Fetch ALL 5 latent vectors
```

With DSA:

```
Fetch only 3 latent vectors
```

At:

- 5 tokens → small savings

At:

- 128K tokens → enormous savings

That is the entire purpose of DSA.

My Personal Summary

The "Evil" of KV Cache (Quick Revision)

While KV cache speeds up computation, it creates a massive memory bottleneck.

- **Storage:** KV cache must reside in **High Bandwidth Memory (HBM/VRAM)**.
- **Impact:** As sequence length (S) and batch size (B) grow, the cache size grows linearly.
- **GPU Roofline Model:**
    - $\text{Arithmetic Intensity} = \frac{\text{FLOPs}}{\text{Bytes Transferred}}$.
    - Large KV cache increases "Bytes Transferred," lowering arithmetic intensity.
    - This shifts the GPU operation into the **Memory Bound** region (left side of the roofline), limiting speed regardless of compute power.

Goal: Reduce the size of the KV cache to increase arithmetic intensity and move operations toward the **Compute Bound** region.
KV Cache Memory Formula:
$\text{Memory} = L \times B \times N \times S \times H \times 2 \times 2$

- L: Number of Transformer Layers
- B: Batch Size
- N: Number of Attention Heads
- S: Sequence Length
- H: Head Dimension
- 2: For Keys and Values
- 2: Bytes per parameter (assuming FP16)

Multi-Head Attention (MHA) - The Baseline

- **Mechanism:** Each head has its own unique trainable weight matrices for **Query** (W_Q), **Key** (W_K), and **Value** (W_V).
- **Result:** N distinct attention heads, each capturing different perspectives (e.g., syntax, semantics, coreference).
- **Memory Cost:** Highest. Stores N unique K and V matrices.
- **Performance:** Best accuracy/perplexity, but slow inference due to memory bandwidth limits.

Multi-Query Attention (MQA) - The "Lazy" Extreme

- **Mechanism:** All heads share the **same** Key and Value weight matrices (W_K, W_V). Only the Query weights (W_Q) remain distinct per head.
- **Result:**
  - $K_1 = K_2 = \cdots = K_N$
  - $V_1 = V_2 = \cdots = V_N$
  - Queries ($Q_1 \ldots Q_N$) are still distinct, so attention scores differ slightly, but the perspective diversity is significantly reduced.
- **Memory Reduction:** Reduces the N term to **1**.
  $\text{Memory MQA} \approx \frac{1}{N} \times \text{Memory MHA}$
- **Trade-off:** Massive speedup and memory savings, but noticeable drop in model quality (perplexity).
- **Usage:** Rarely used in modern high-quality models (e.g., Google PaLM used it, but it's largely abandoned for better variants).

Grouped-Query Attention (GQA) - The Sweet Spot

- **Mechanism:** A compromise between MHA and MQA.
  - Heads are divided into G **groups**.
  - All heads within a group share the same K and V matrices.
  - Different groups have different K and V matrices.
- **Result:**
  - Reduces the N term to G (Number of Groups)
    $\text{Memory GQA} \approx \frac{G}{N} \times \text{Memory MHA}$
- **Example:** Llama 3 uses 32 query heads but only 8 KV heads (groups).
  $Ratio = 4 : 1$
- **Trade-off:**
  - **Memory:** Significant reduction (e.g., $4x$ reduction for Llama).
  - **Quality:** Minimal loss in perplexity compared to MHA.
- **Adoption:** Widely used in modern open-source models (Llama 3, Gemma, Mistral).

Multi-Head Latent Attention (MLA) - The "DeepSeek" Innovation

- **Concept:** Instead of reducing the number of heads (which loses information), **compress the information** itself.
- **Mechanism:**
    1. **Down Projection:** Input is projected into a low-dimensional **Latent Space** (D_{latent}) using W_{DKV}.
    2. **Caching:** Only the small **Latent Vector** (C_{KV}) is cached, not the full K and V.
    3. **Up Projection:** When attention is needed, C_{KV} is projected back to full dimensions using W_{UK} and W_{UV} to reconstruct K and V.
- **Memory Formula:**
  $\text{Memory MLA} = L \times B \times S \times D_{latent} \times 2$
    - Replaces $N \times H$ with D_{latent}.
- **DeepSeek Example:**
    - Original Embedding: $16,384$
    - Latent Dimension: 512
    - **Reduction:** $\approx 64\times$ reduction in KV cache size.
- **Advantages:**
    - **Best of Both Worlds:** Maintains full head diversity (unlike MQA/GQA) because up-projection matrices are distinct per head.
    - **Massive Memory Savings:** Drastic reduction in bytes transferred.
- **Performance:** High throughput with near-MHA quality. Used in DeepSeek V3/V4, Kimi, and GLM.

DeepSeek Sparse Attention (DSA)

Introduced in DeepSeek V3.2 and used in V4.

1. The Problem with Long Contexts

Even with MLA, if the sequence length (S) is 128k, the model must read $128,000$ latent vectors from HBM for every new token. This is still memory-intensive.

2. The Solution: Selective Reading

DSA introduces a mechanism to read only the **most relevant** tokens ($Top-K$) instead of the entire context.

3. Mechanism

1. **Indexer Vectors:** For every stored token, a small, low-dimensional **Index Key** vector is cached alongside the Latent Vector.
2. **Lookup Query:** When a new token arrives, it generates a **Lookup Query** vector.
3. **Scoring:** The Lookup Query is dot-producted with all cached Index Keys to generate "relevance scores."
4. **Selection:** Only the top K tokens (e.g., 2048 out of $128,000$) with the highest scores are selected.
5. **Reading:** Only the Latent Vectors for these K tokens are fetched from HBM to compute attention.

4. Benefits

- **Memory Transfer Reduction:** Instead of reading S vectors, the model reads only K vectors (+ a small cost for the index scan).
- **Roofline Impact:** Significantly increases arithmetic intensity by reducing bytes transferred.
- **Comparison to Sliding Window:**
  - *Sliding Window:* Fixed window (e.g., last $4k$ tokens). Ignores important tokens far in the past.
  - *Sparse Attention:* **Dynamic selection.** Can retrieve relevant tokens from anywhere in the $128k$ context.

Comparative Summary

Mechanism	KV Cache Reduction	Head Diversity	Quality (Perplexity)	Inference Speed	Key Trade-off
MHA	None (Baseline)	Full	Highest	Slowest	High memory usage
MQA	$1/N$	Low (Shared K/V)	Low	Fastest	Significant quality drop
GQA	G/N	Medium (Grouped)	High	Fast	Requires tuning group size
MLA	$\frac{D_{latent}}{(N \times H)}$	Full	High (Near MHA)	Very Fast	Slight compute overhead for projection
DSA	Read K of S	Full	High	Fastest (Long Context)	Complexity of indexing logic