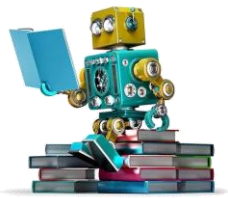
SDML
ML Paper Review

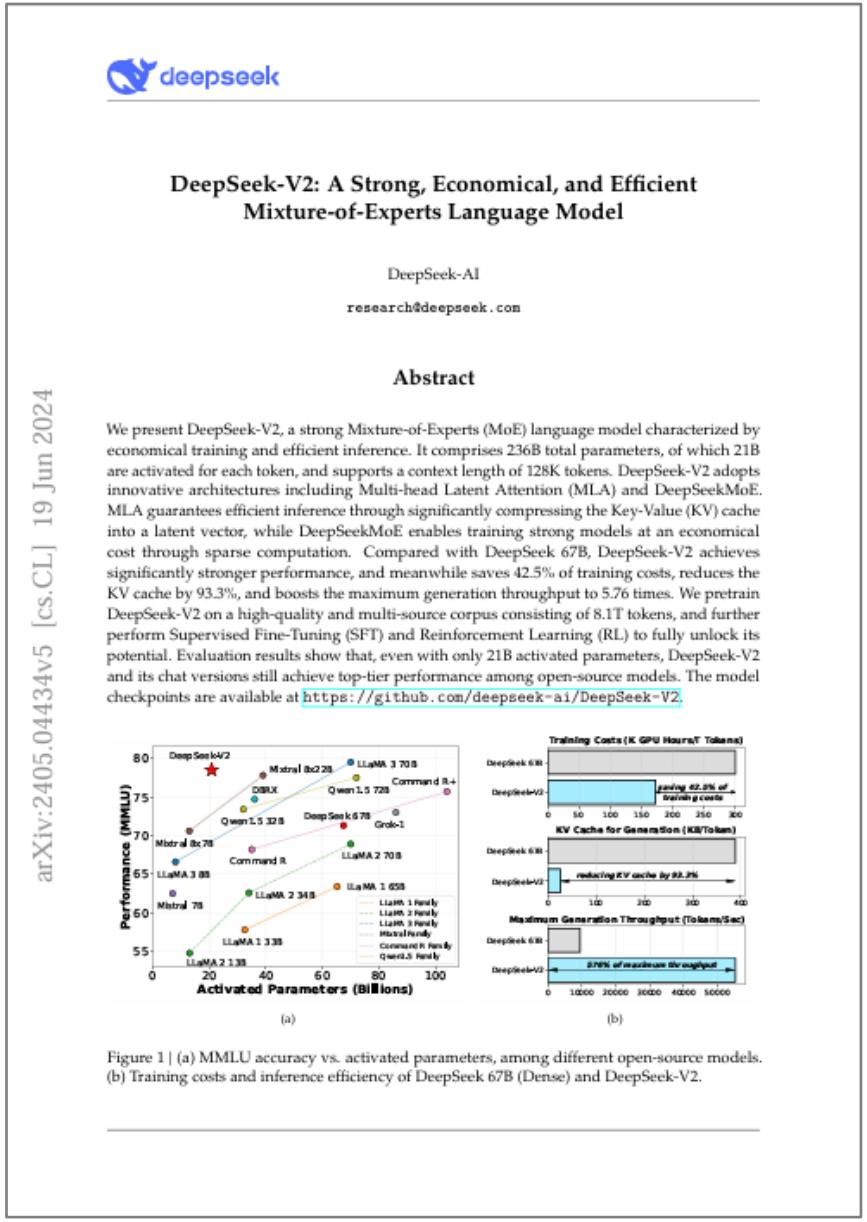
November 2024



Multi-head Latent Attention (from DeepSeek-V2)

DeepSeek-AI

https://arxiv.org/abs/2405.04434



deepseek

DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model

DeepSeek-AI

research@deepseek.com

Abstract

We present DeepSeek-V2, a strong Mixture-of-Experts (MoE) language model characterized by economical training and efficient inference. It comprises 236B total parameters, of which 21B are activated for each token, and supports a context length of 128K tokens. DeepSeek-V2 adopts innovative architectures including Multi-head Latent Attention (MLA) and DeepSeekMoE. MLA guarantees efficient inference through significantly compressing the Key-Value (KV) cache into a latent vector, while DeepSeekMoE enables training strong models at an economical cost through sparse computation. Compared with DeepSeek 67B, DeepSeek-V2 achieves significantly stronger performance, and meanwhile saves 42.5% of training costs, reduces the KV cache by 93.3%, and boosts the maximum generation throughput to 5.76 times. We pretrain DeepSeek-V2 on a high-quality and multi-source corpus consisting of 8.1T tokens, and further perform Supervised Fine-Tuning (SFT) and Reinforcement Learning (RL) to fully unlock its potential. Evaluation results show that, even with only 21B activated parameters, DeepSeek-V2 and its chat versions still achieve top-tier performance among open-source models. The model checkpoints are available at https://github.com/deepseek-ai/DeepSeek-V2.

Figure 1 | (a) MMLU accuracy vs. activated parameters, among different open-source models. (b) Training costs and inference efficiency of DeepSeek 67B (Dense) and DeepSeek-V2.

DeepSeek-V2 overview

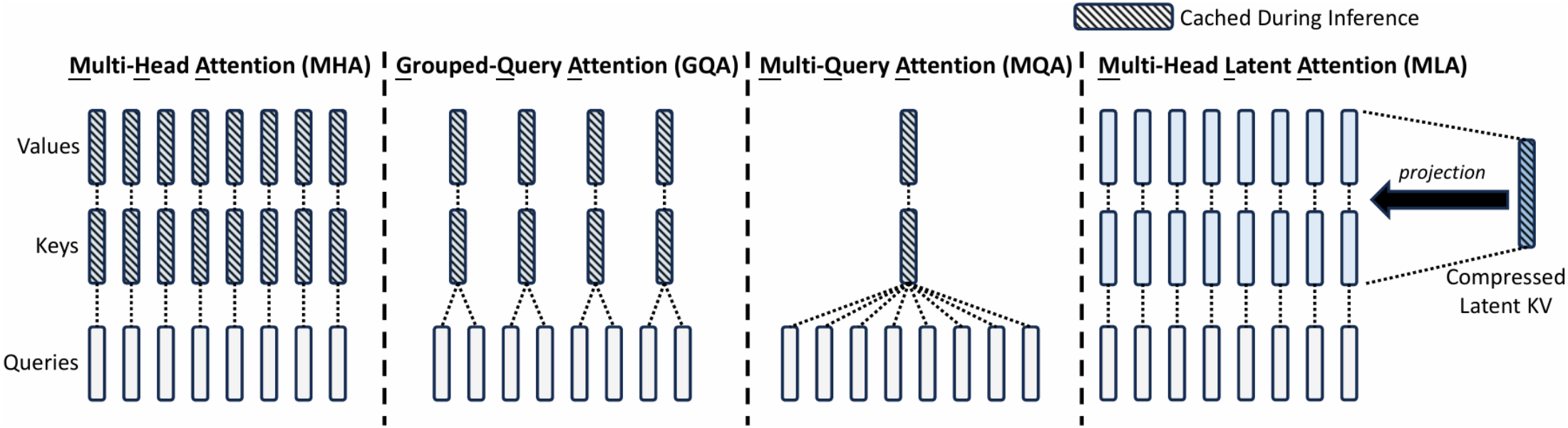
- DeepSeek-V2 is an open source LLM from the company DeepSeek AI
- The paper is recent, dated June 2024, and it includes many of the latest LLM design and architecture choices, including:
  - Very large total parameter count at 236B parameters
  - Trained on a large 8.1T token corpus
  - Heavy Mixture of Experts use, with only 21B parameters used for each token
  - Long context length of 128K tokens
  - Not only uses a KV cache during inference, but also implements multi-head latent attention (MLA), the subject of today's talk
- MLA is a key reason why DeepSeek-V2 uses a KV cache that is 15x smaller and has 5.76 times the throughput of the first version

Multi-head latent attention overview

- For inference with long contexts, using a KV cache is faster than recomputing keys and values for the context
- However, the time to load cached keys & values is the bottleneck
- Multi-head latent attention (MLA) saves a much smaller latent than the raw keys & values, reducing the time to load significantly
- There is extra computation for reconstruction, but many recent optimizations, such as FlashAttention, have demonstrated that maximizing GPU performance is more about reducing memory access than about reducing FLOPS

Prior approaches to reduce size of KV cache

- All approaches use as many queries as there are tokens
  - Multi-Query Attention (MQA) uses only 1 key & value shared by all heads
  - Grouped-Query Attention (GQA) uses 1 key & value for every n (e.g., 2 or 4) heads



Performance impact of GQA and MQA

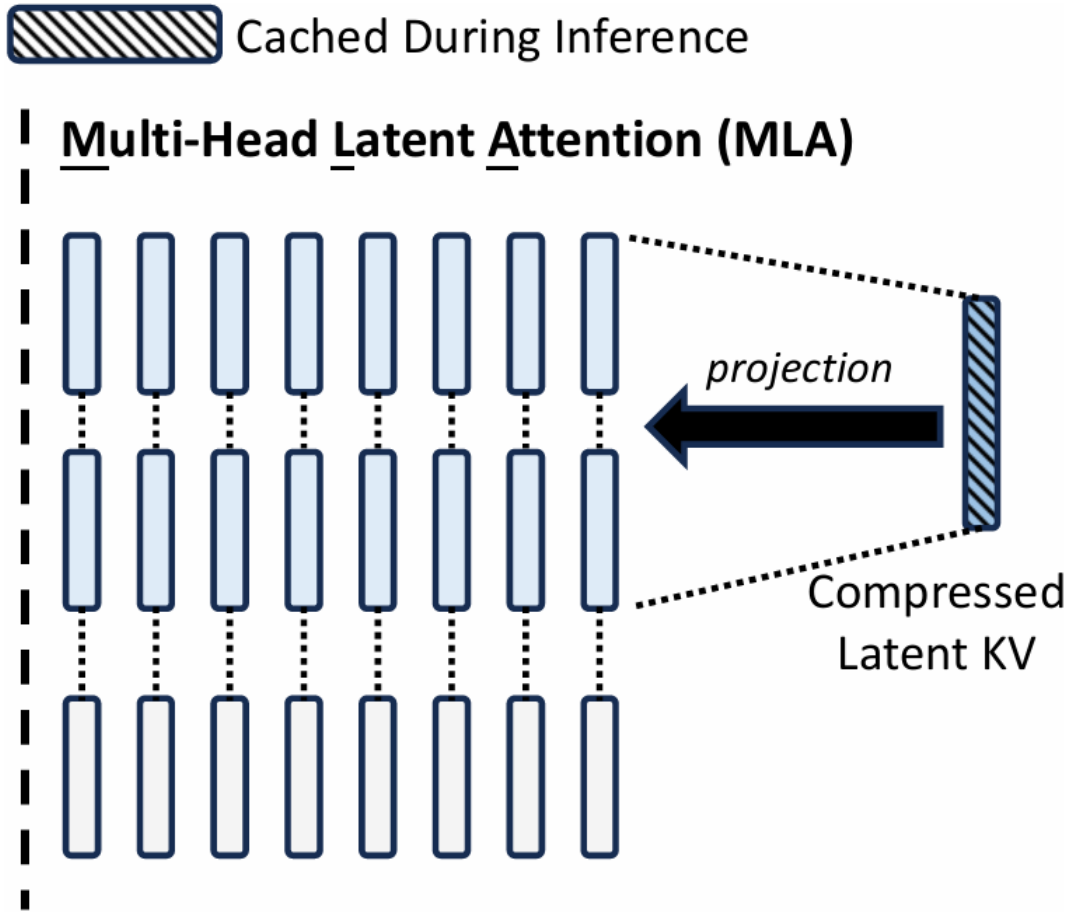
- While reducing the size of the KV cache, GQA and MQA negatively impact LLM performance

Benchmark (Metric)	# Shots	Dense 7B w/ MQA	Dense 7B w/ GQA (8 Groups)	Dense 7B w/ MHA
# Params	-	7.1B	6.9B	6.9B
BBH (EM)	3-shot	33.2	35.6	**37.0**
MMLU (Acc.)	5-shot	37.9	41.2	**45.2**
C-Eval (Acc.)	5-shot	30.0	37.7	**42.9**
CMMLU (Acc.)	5-shot	34.6	38.4	**43.5**

Table 8 | Comparison among 7B dense models with MHA, GQA, and MQA, respectively. MHA demonstrates significant advantages over GQA and MQA on hard benchmarks.

Multi-head latent attention method

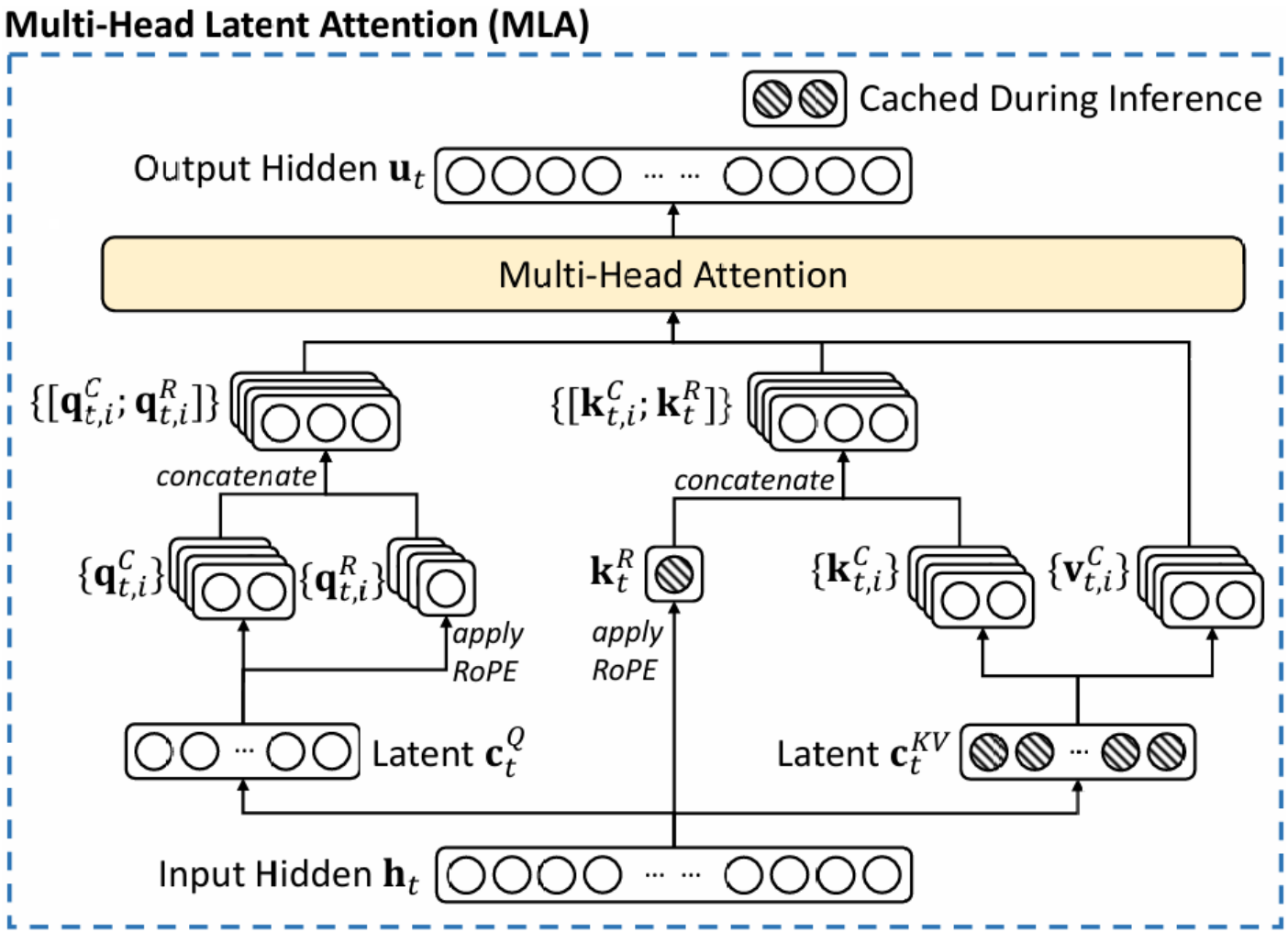
- Since reducing the number of keys & values doesn't seem to work, keep all of them, but compress them
- What form of compression?
- Simply use a matrix multiply to reduce them to a much lower dimension
  - In the paper, they say they reduce to about 7% original size



MLA and RoPE

- Rotary position embeddings (RoPE) are very popular (e.g., Llama)
- Standard RoPE applies rotation matrices to the queries and keys, with the intuition that the difference in rotation is solely dependent on the relative token distance, not the absolute token number
- But applying RoPE rotations to the keys would make it harder to learn good compression of the keys
- MLA's solution is to have small RoPE vectors for queries and keys that are concatenated with the main queries and keys

Overall MLA calculation



MLA storage reduction

Attention Mechanism	KV Cache per Token (# Element)	Capability
Multi-Head Attention (MHA)	$2n_h d_h l$	Strong
Grouped-Query Attention (GQA)	$2n_g d_h l$	Moderate
Multi-Query Attention (MQA)	$2d_h l$	Weak
MLA (Ours)	$(d_c + d_h^R)l \approx \frac{9}{2} d_h l$	Stronger

Table 1 | Comparison of the KV cache per token among different attention mechanisms. n_h denotes the number of attention heads, d_h denotes the dimension per attention head, l denotes the number of layers, n_g denotes the number of groups in GQA, and d_c and d_h^R denote the KV compression dimension and the per-head dimension of the decoupled queries and key in MLA, respectively. The amount of KV cache is measured by the number of elements, regardless of the storage precision. For DeepSeek-V2, d_c is set to $4d_h$ and d_h^R is set to $\frac{d_h}{2}$. So, its KV cache is equal to GQA with only 2.25 groups, but its performance is stronger than MHA.

MLA performance

- Interestingly, DeepSeek **claims MLA outperformed regular attention**!
  - It's appropriate to have some skepticism about paper results

Benchmark (Metric)	# Shots	Small MoE w/ MHA	Small MoE w/ MLA	Large MoE w/ MHA	Large MoE w/ MLA
# Activated Params	-	2.5B	2.4B	25.0B	21.5B
# Total Params	-	15.8B	15.7B	250.8B	247.4B
KV Cache per Token (# Element)	-	110.6K	15.6K	860.2K	34.6K
BBH (EM)	3-shot	37.9	**39.0**	46.6	**50.7**
MMLU (Acc.)	5-shot	48.7	**50.0**	57.5	**59.0**
C-Eval (Acc.)	5-shot	**51.6**	50.9	57.9	**59.2**
CMMLU (Acc.)	5-shot	52.3	**53.4**	60.7	**62.5**

Table 9 | Comparison between MLA and MHA on hard benchmarks. DeepSeek-V2 shows better performance than MHA, but requires a significantly smaller amount of KV cache.

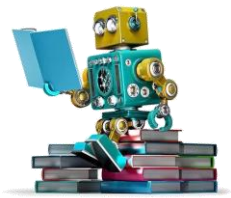
Multi-head latent attention conclusion

- MLA reduced the KV cache size dramatically, without suffering the same performance penalty of grouped-query attention (GQA)
  - In fact, reported better performance than regular multi-headed attention
- The vanilla formulation involves a matrix multiply to a lower rank, shorter vector representation – about 7% size for DeepSeek-V2
  - A little extra work is needed to make RoPE position embeddings work
  - Performance optimizations for inference allow them to combine/precompute two pairs of matrix multiplies, so that the extra compute is minimal
- The DeepSeek-V2 has other optimizations with MoE and training
- DeepSeekV2 uses only 21B parameters during inference and claims similar performance to Qwen1.5 72B, Mixtral 8x22B, and Llama 3 70B

References

- DeepSeek-V2 model checkpoints
  https://github.com/deepseek-ai/DeepSeek-V2
- DeepSeek LLM: Scaling Open-Source Language Models with Longtermism
  Xiao Bi et al. (2024)
  https://arxiv.org/abs/2401.02954
- DeepSeekMoE: Towards Ultimate Expert Specialization in Mixture-of-Experts Language Models
  Damai Dai et al. (2024)
  https://arxiv.org/abs/2401.06066

Appendix



Full MLA formulas

- WiM is ir ext

$$\mathbf{c}_t^Q = W^{DQ}\mathbf{h}_t, \tag{37}$$

$$[\mathbf{q}_{t,1}^C; \mathbf{q}_{t,2}^C; \ldots; \mathbf{q}_{t,n_h}^C] = \mathbf{q}_t^C = W^{UQ}\mathbf{c}_t^Q, \tag{38}$$

$$[\mathbf{q}_{t,1}^R; \mathbf{q}_{t,2}^R; \ldots; \mathbf{q}_{t,n_h}^R] = \mathbf{q}_t^R = \mathrm{RoPE}(W^{QR}\mathbf{c}_t^Q), \tag{39}$$

$$\mathbf{q}_{t,i} = [\mathbf{q}_{t,i}^C; \mathbf{q}_{t,i}^R], \tag{40}$$

$$\boxed{\mathbf{c}_t^{KV}} = W^{DKV}\mathbf{h}_t, \tag{41}$$

$$[\mathbf{k}_{t,1}^C; \mathbf{k}_{t,2}^C; \ldots; \mathbf{k}_{t,n_h}^C] = \mathbf{k}_t^C = W^{UK}\mathbf{c}_t^{KV}, \tag{42}$$

$$\boxed{\mathbf{k}_t^R} = \mathrm{RoPE}(W^{KR}\mathbf{h}_t), \tag{43}$$

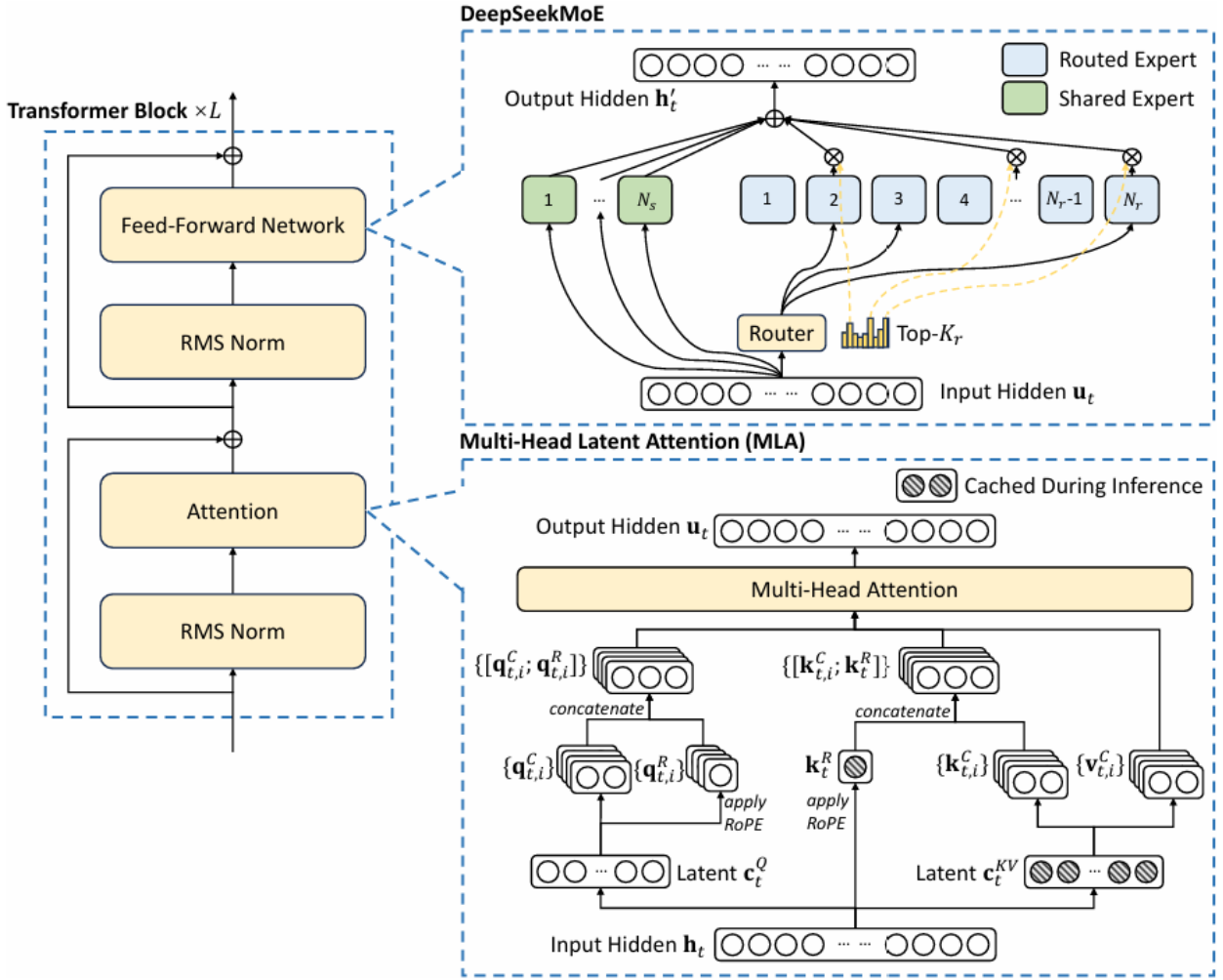
$$\mathbf{k}_{t,i} = [\mathbf{k}_{t,i}^C; \mathbf{k}_t^R], \tag{44}$$

$$[\mathbf{v}_{t,1}^C; \mathbf{v}_{t,2}^C; \ldots; \mathbf{v}_{t,n_h}^C] = \mathbf{v}_t^C = W^{UV}\mathbf{c}_t^{KV}, \tag{45}$$

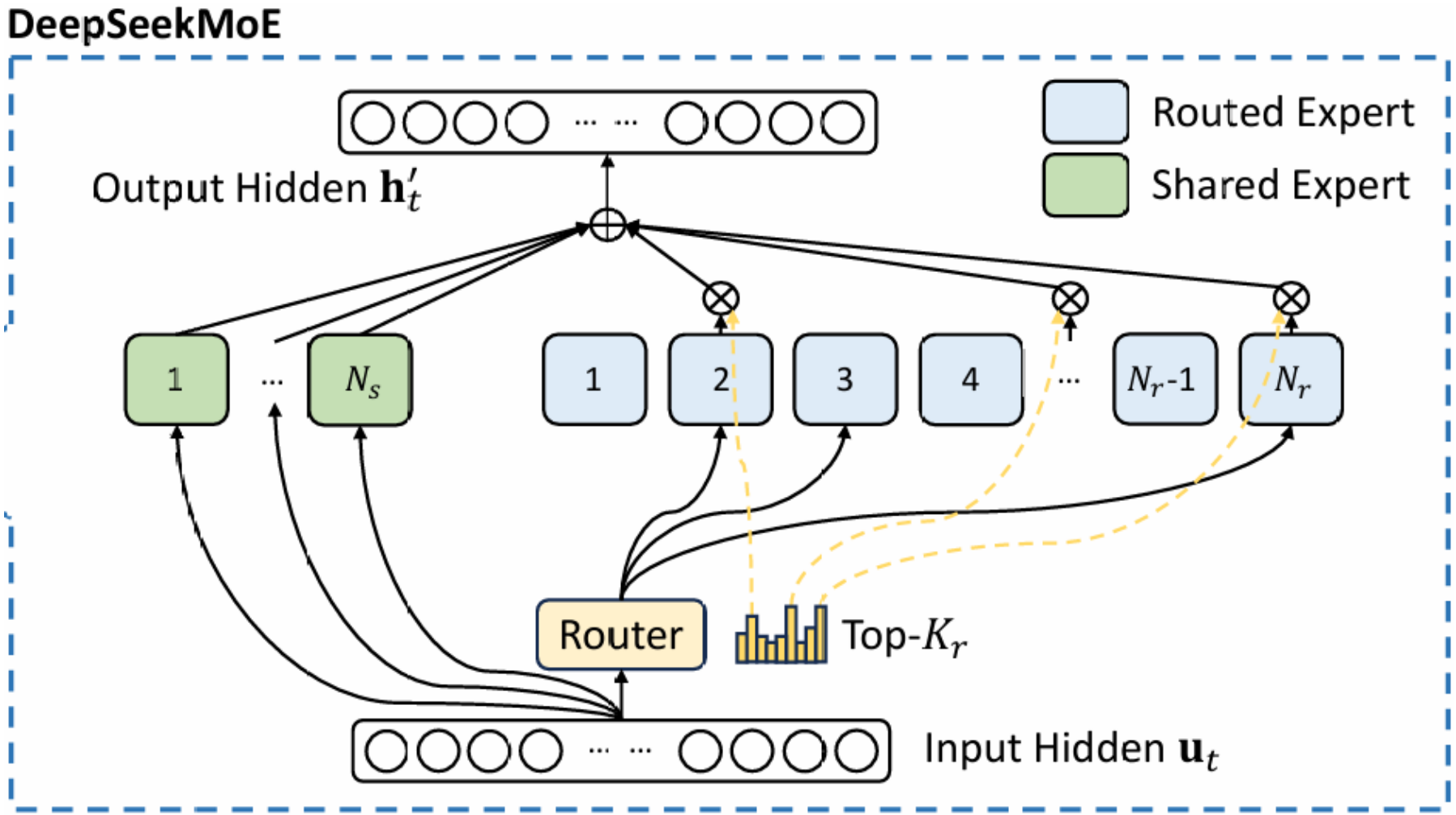
$$\mathbf{o}_{t,i} = \sum_{j=1}^{t} \mathrm{Softmax}_j\left(\frac{\mathbf{q}_{t,i}^T \mathbf{k}_{j,i}}{\sqrt{d_h + d_h^R}}\right)\mathbf{v}_{j,i}^C, \tag{46}$$

$$\mathbf{u}_t = W^O[\mathbf{o}_{t,1}; \mathbf{o}_{t,2}; \ldots; \mathbf{o}_{t,n_h}], \tag{47}$$

DeepSeek-V2 overall architecture

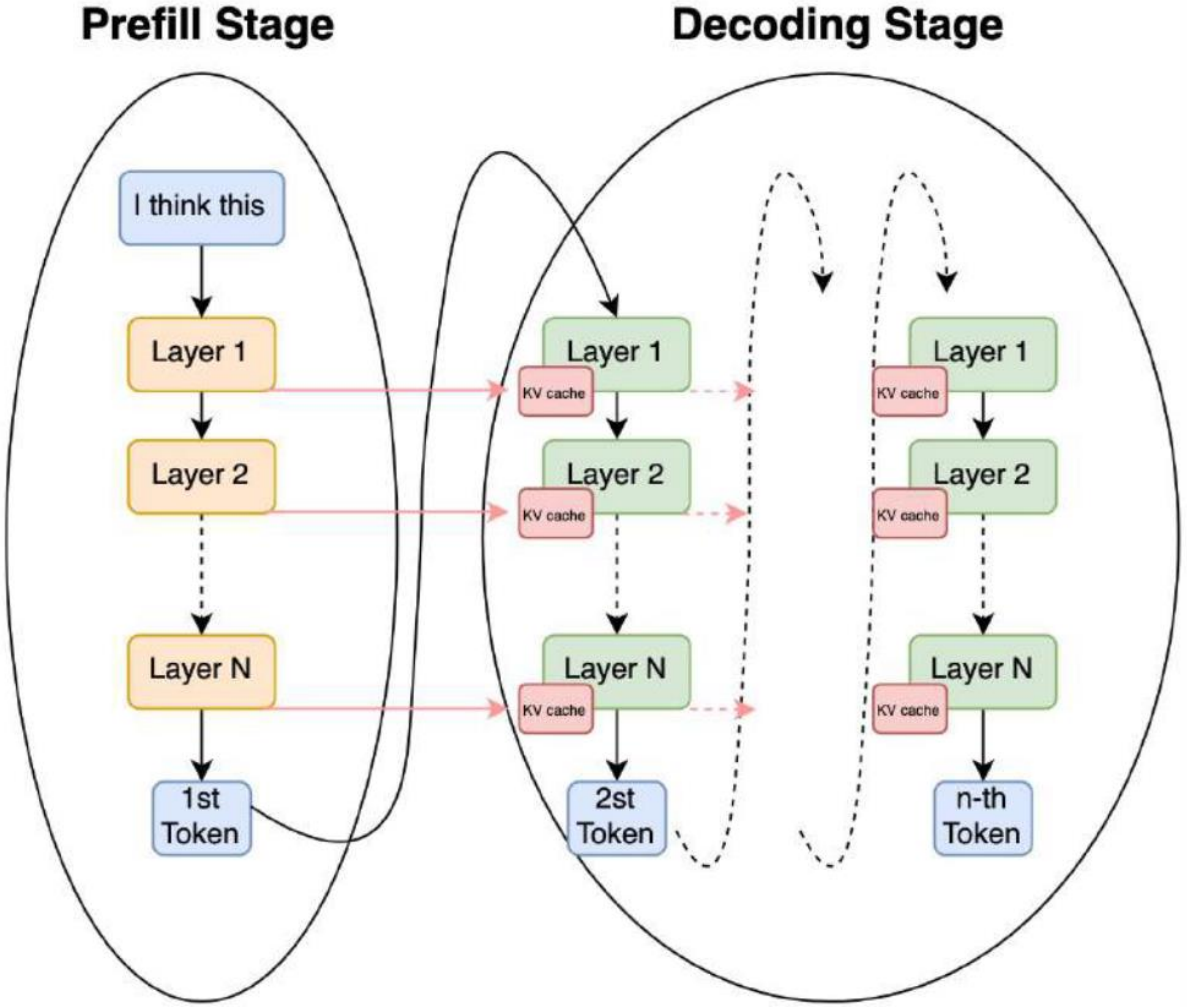


DeepSeek-V2 MoE



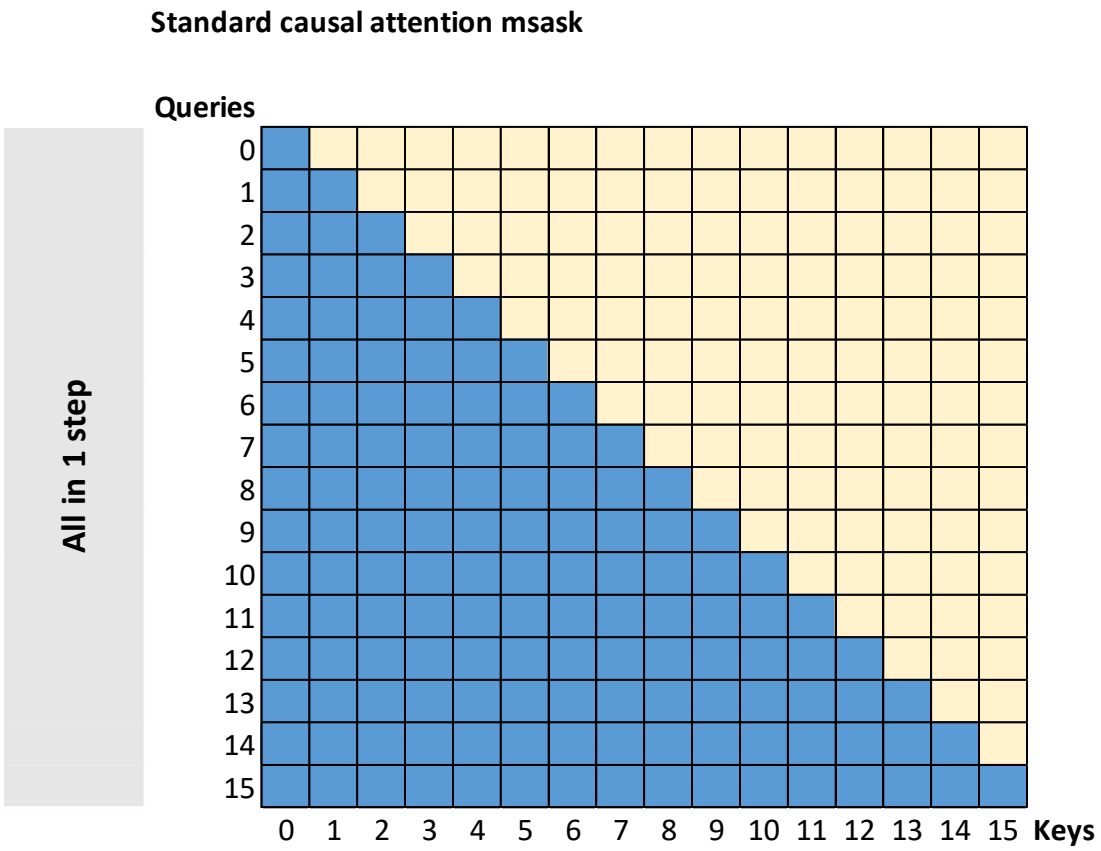
Prefill and decoding stages

- LLM inference happens in two very different stages
- During the prefill stage, we have many tokens from the prompt we can process together
- During decoding, we predict the next token, one token at a time



https://sites.google.com/view/kdd-2023-tutorial/home

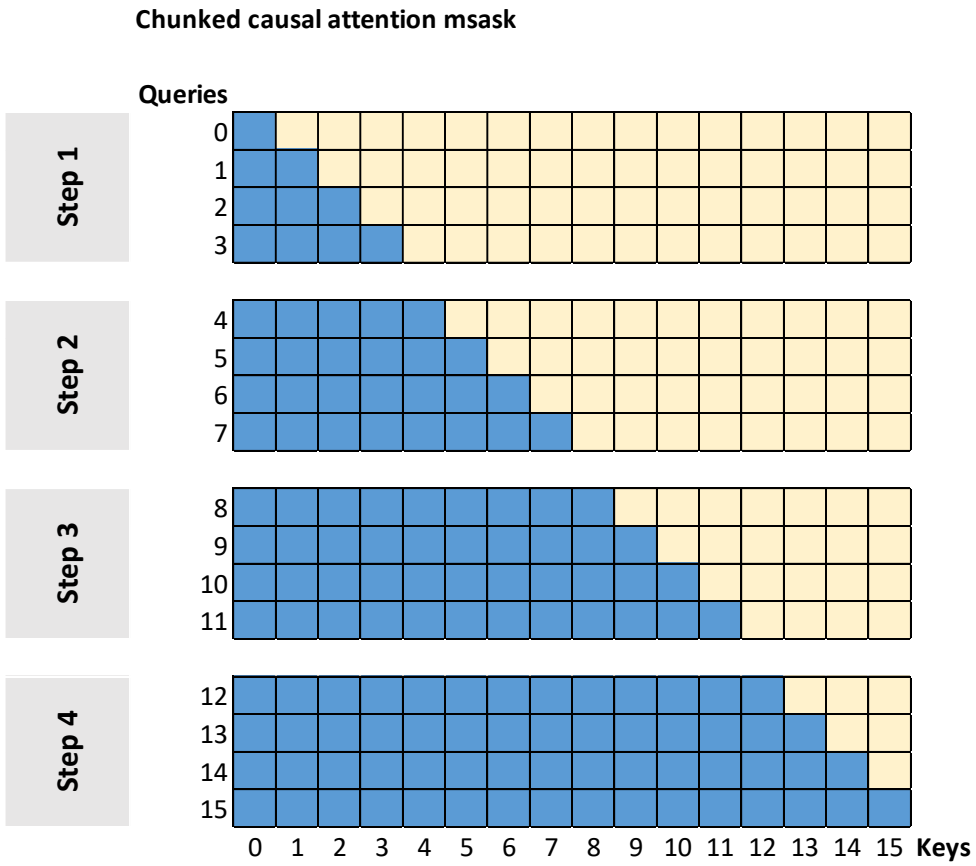
Standard attention



Blue squares are keys that each query is allowed to attend to
Yellow squares will have zero attention scores

- Every query, shown in rows, is multiplied with every key, shown in columns
- Keep only the blue squares
- In each position, the same query vector is multiplied with every one of the keys
- Likewise the same key vector values will be used with every one of the queries

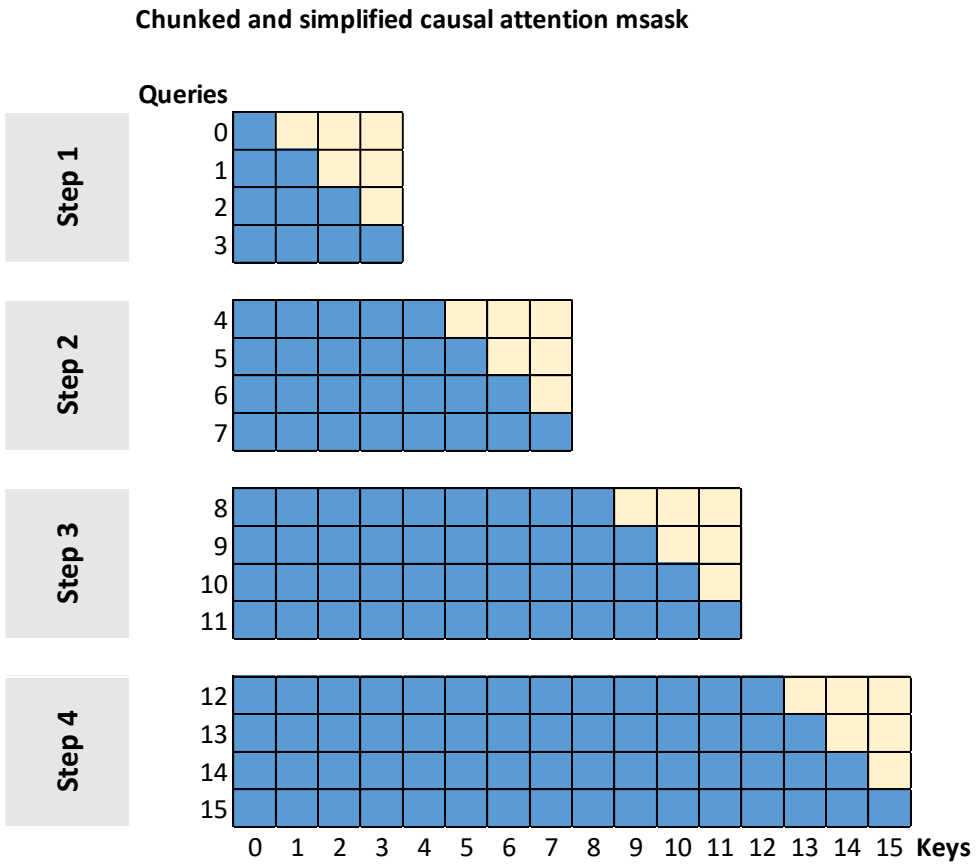
Chunked attention



Blue squares are keys that each query is allowed to attend to
Yellow squares will have zero attention scores

- Breaking the attention calc into chunks reduces size of each step
- It’s the same number of FLOPS
- But this reduces memory needed
- Concatenating the results of all steps results in the exact same answer

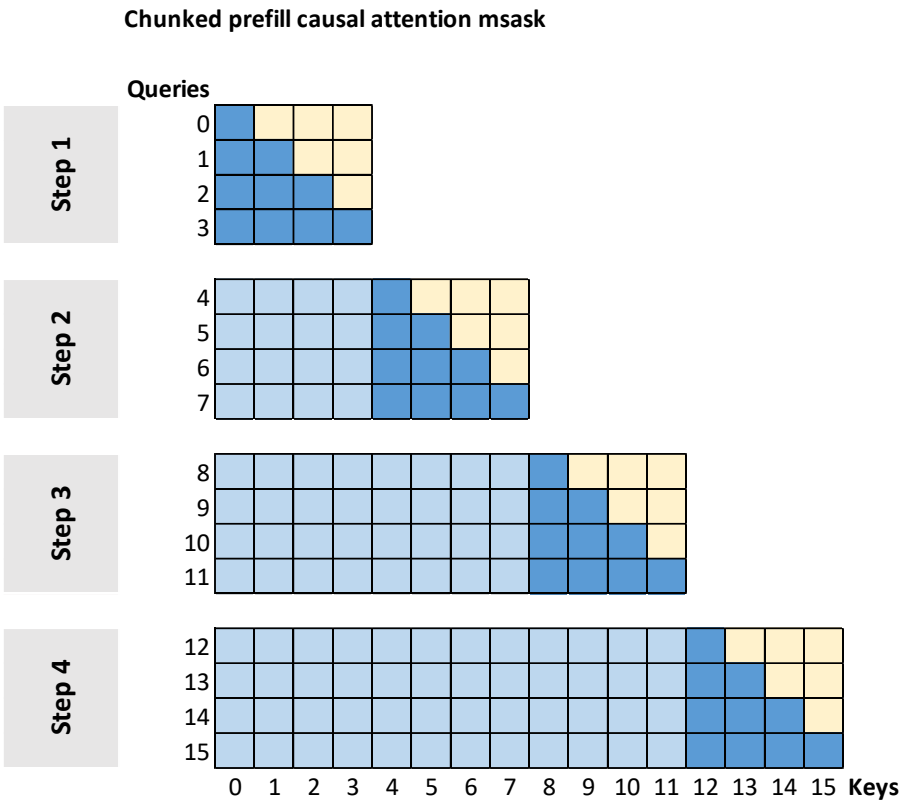
Simplified chunked attention



Blue squares are keys that each query is allowed to attend to
Yellow squares will have zero attention scores

- Since we are throwing away the results in yellow squares, we don't have to actually calculate them
- We can do these smaller matrix multiplies
- When concatenating the results, we need to pad zeros wherever we removed yellow squares

Chunked prefill attention



Blue squares are keys that each query is allowed to attend to

Dark blue squares calculate keys and values from scratch, and those are used in attention

Light blue squares reuse cached keys and values from previous steps, and those are used in attention

Yellow squares will have zero attention scores

- If we do the steps in order, from top to bottom, then the keys and values in light blue will have already been calculated by earlier steps
- We can cache and reuse them
- More importantly, we can continue to cache keys and values when decoding proceeds

Chunked Prefill

- Typical LLM inference consists of prefill phase then decoding phase
  - Prefill computes all keys and values for the prompt, storing in the KV cache
  - Decoding computes one next token at a time, using KV cache
- Naïve prefill of a long context of length L:
  - Performs attention on L queries and L keys, requiring $O(L^2)$ memory
  - It also can tie up the GPU for a long period of time, forcing other users to wait until the entire computation is finished
- Dividing up the prefill phase, called *chunked prefill* improves both:
  - Say you have N chunks of length K
  - Each chunk of K queries requires $O(KL)$ memory
  - After each chunk finishes, they system can decide if it wants to combine other users into the same batch as the next chunk, lowering response time