Kimi K3, The Manos, The Mythos, The Legendos

Kimi K3, The Manos, The Mythos, The Legendos Kimi K3's architecture: compressed memory, attention across depth, latent expert routing, and the inference performance

AUG 3 https://newsletter.semianalysis.com/p/kimi-k3-the-manos-the-mythos-the

Kimi K3 took the world by storm at its announcement, sweeping leaderboards and establishing itself as the open frontier model. While the community is eager to understand how Kimi K3 works, many have been surprised by the unconventional techniques driving its performance. This article serves as a primer to understanding the core techniques of the Kimi K3 model architecture.

QQ:1413221435

Kimi Delta Attention

Kimi Delta Attention (KDA) is the linear attention layer in Kimi K3's hybrid attention mechanism. We trace the origins of KDA, starting from linear attention, DeltaNet, Gated DeltaNet (GDN), then to KDA.

Linear Attention

The derivation of linear attention stems from **removing the softmax operation in the standard softmax attention**. Below we compare the iterative inference formulas, which show the computation of the output vector at token position t:

By removing the softmax operation, we can reorder the operations and reduce the computation complexity of attention from quadratic to linear:

QQ:1413221435

The new equations are as follows:

QQ:1413221435

Vectors q, k, v, have dimensions *L* by *d*. The computational complexity of both equations are O(Ld^2), thereby making the computation linear. Comparing the new equations with softmax attention's equation, we see that **softmax attention requires accessing all past key and value vectors**, whereas **linear attention compresses all past key and value vectors into one hidden state S**.

We reinterpret the new equations as an **online learning objective**. We view matrix S as an associative memory that stores the associations between key vector k and value vector v, and we retrieve v by multiplying S with k. We can then interpret the first equation as **continuously updating the matrix S at every position to perfect the retrieval**. Finally, we can interpret the vt @ kt.T term as the gradient of loss function -(S @ kt.T) @ vt with respect to S.

$$\textbf{Objective:} \quad \mathcal{L}_t(\mathbf{S}) = -\langle \mathbf{S}\mathbf{k}_t, \mathbf{v}_t \rangle$$

$$\begin{aligned} \textbf{SGD update:} \quad \mathbf{S}_t &= \mathbf{S}_{t-1} - \beta_t \nabla \mathcal{L}_t(\mathbf{S}_{t-1}) \\ &= \mathbf{S}_{t-1} + \beta_t \mathbf{v}_t \mathbf{k}_t^\top \end{aligned}$$

DeltaNet

QQ:1413221435

Under the online learning objective view, we see the values of matrix S will grow unboundedly: old and new information gets blurred together in S as the sequence grows, which destabilizes learning. Without softmax giving well-scaled and bounded outputs, linear attention typically lags behind softmax attention on long-range recall tasks.

DeltaNet improves upon linear attention by **changing the loss function to minimizing the L2 norm of the value retrieval**. Unlike linear attention's loss function, DeltaNet's loss function regularizes the growth of S. This creates a new matrix S update rule, the Delta Rule, as below:

$$\textbf{Objective:} \quad \mathcal{L}_t(\mathbf{S}) = \frac{1}{2}\|\mathbf{S}\mathbf{k}_t - \mathbf{v}_t\|^2$$

$$\begin{aligned}\textbf{SGD update:} \quad \mathbf{S}_t &= \mathbf{S}_{t-1} - \beta_t \nabla \mathcal{L}_t(\mathbf{S}_{t-1}) \\ &= \mathbf{S}_{t-1} - \beta_t \left(\mathbf{S}_{t-1}\mathbf{k}_t - \mathbf{v}_t\right)\mathbf{k}_t^\top\end{aligned}$$

Source: Linear Attention and Beyond (Interactive Tutorial with Songlin Yang)

The Delta Rule becomes the basis of DeltaNet's attention equation:

$$\mathbf{S}_t = \mathbf{S}_{t-1} - \beta_t \left(\mathbf{S}_{t-1}\mathbf{k}_t - \mathbf{v}_t\right)\mathbf{k}_t^\top$$

Conceptually, $S_{t-1} @ k_t - v_t$ represents the associations irrelevant to the current key and value, and DeltaNet performs targeted removal of those associations.

Gated DeltaNet

GDN and KDA are adaptations of DeltaNet. Gated DeltaNet applies the LSTM forget gate *alpha* on the matrix S, allowing the model to control memory lifespan with weight decay. KDA further expands *alpha* into a diagonal matrix that enables fine-grained per-channel memory decay and positional awareness.

QQ:1413221435
QQ:1413221435

Thanks for reading SemiAnalysis! This
post is public so feel free to share it.

FlashKDA Algorithm

Moonshot developed FlashKDA, their custom kernels for KDA, and open-sourced it. Here we explain the algorithm and derive the arithmetic intensity.

Algorithm

First, let's start from an alternative formulation of the recurrence formula:

```
u_t = beta_t * (v_t - (D_t @ S_t-1).T @ k_t)
S_t = D_t @ S_t-1 + k_t @ u_t.T
o_t.T = q_t.T @ S_t
```

QQ:1413221435

Here, D_t is the diagonal matrix of the alpha forget gate, and u_t is the delta in the delta rule. For decode, the kernel roughly follows the formula. For prefill, we parallelize the operation by unrolling the recurrence formula in chunks of tokens, in order to efficiently execute the operations on GPUs. Assume we unroll token i to j, and the starting state is S_i-1, we get:

```
S_j = D_j:i @ S_i-1 + sum(D_j:t+1 @ k_t @ u_t.T, t=i:j)
o_j.T = q_j.T @ S_j
      = q_j.T @ D_j:i @ S_i-1 + sum(q_j.T @ D_j:t+1 @ k_t @ u_t.T, t=i:j)
```

D_j:i refers to the cumulative decay from token i to j: D_j @ D_j-1 @ D_j-2 @ … @ D_i. In FlashKDA's matrix form, the formula becomes:

```
S_out = D_j:i @ S_in + K_restore.T @ U
M_qk = tril(Q_decay @ K_inv.T)
O = Q_decay @ S_in + M_qk @ U
```

The vector to matrix mapping is as follows:

QQ:1413221435

- S_in refers to the state at the starting position of a chunk
- S_out refers to the state at the end position of a chunk

- K_restore is the matrix form of D_j:t+1 @ k_t
- Q_decay is the matrix form of q_j.T @ D_j:i
- Q_decay @ K_inv.T the matrix form of q_j.T @ D_j:t+1 @ k_t, derived from (q_j.T @ D_j:i) @ (D_t:i^-1 @ k_t)
- M_qk is the causal mask, so it's a lower triangular matrix

U is the matrix form of unrolled u_t. To compute this, we apply UT transform and compute the following:

```
B = Diag(beta) @ (V - K_decay @ S_in)
L = StrictTril(Diag(beta) @ K_decay @ K_inv.T)
U = (I + L)^-1 @ B
```

QQ:1413221435

Please consult Songlin Yang's blog post and Kimi Linear paper section 3.1 for the full derivation. Note that here U corresponds to the pseudo-value term in the Kimi Linear paper.

Implementation-wise, FlashKDA launches 2 kernels: K1 and K2. K1 prepares chunk-level tensors in parallel, including:

```
a = exp2(cumsum(g))
K_decay = Diag(a) @ K
Q_decay = Diag(a) @ Q
K_inv = Diag(a)^-1 @ K
K_restore = a[-1] * K_inv
L = StrictTril(Diag(beta) @ K_decay @ K_inv.T); INV = (I + L)^-1
M_qk = tril(Q_decay @ K_inv.T)
```

Here a is the cumulative decay, where each element is the cumulative decay at a token position.

K2 performs chunk-level recurrent computation:

```
U = INV @ Diag(beta) @ (V - K_decay @ S)
O = Q_decay @ S + M_qk @ U
S = Diag(a[-1]) @ S + K_restore.T @ U
```

QQ:1413221435

Complexity Analysis

Here we analyze the complexity of an attention head. For decode, the critical path computations are:

- D_t @ S_t-1: Element-wise multiplication, D × D
- S_t-1.T @ k_t: D × D × 1
- k_t @ u_t.T: D × 1 × D
- q_t.T @ S_t: 1 D × D

The decode kernel roughly performs $7*D^2$ FLOPs.

Reading and writing the FP32 recurrent state dominates the memory traffic, so the memory traffic is roughly $8*D^2$ bytes.

For prefill, the critical path of K1 is at computing L, INV, and M_qk.

- L: C × D × C
- INV: Neumann factorization, performs 6 C × C × C matrix multiplications
- M_qk: C × D × C

For K2,

- K_decay @ S: C × D × D
- Q_decay @ S: C × D × D
- M_qk @ U: C × C × D
- INV @ B: C × C × D
- K_restore.T @ U: D × C × D

Combining K1 and K2, FlashKDA performs $12*C^3 + 8*C^2*D + 6*C*D^2$ FLOPs. Since we analyzed at the chunk level (chunk size C), assuming sequence length T >> C, the overall FLOPs is $T/C * O(C*D^2) = O(T*D^2)$.

For memory traffic:

- K1 read Q, K, g: C × D
- K1 write and K2 read Q_decay, K_decay, K_restore: C × D
- K1 write and K2 read INV, M_qk: C × C
- K2 read V and write O: C × D
- K2 read and write S once per kernel: D × D

QQ:1413221435
QQ:1413221435

In total, FlashKDA accesses 3 * 2*C*D + 2 * (3 * 2*C*D + 2 * 2*C*C) + 2 * 2*C*D = 8*C² + 22*C*D bytes. At the kernel level, it accesses T/C * (8*C² + 22*C*D) + 8*D² ~ O(TC + TD + D²).

This concretely shows that the computational complexity of KDA:

- Prefill: Linear to sequence length for both computation and memory
- Decode: Constant to sequence length for both computation and memory

Kimi Linear

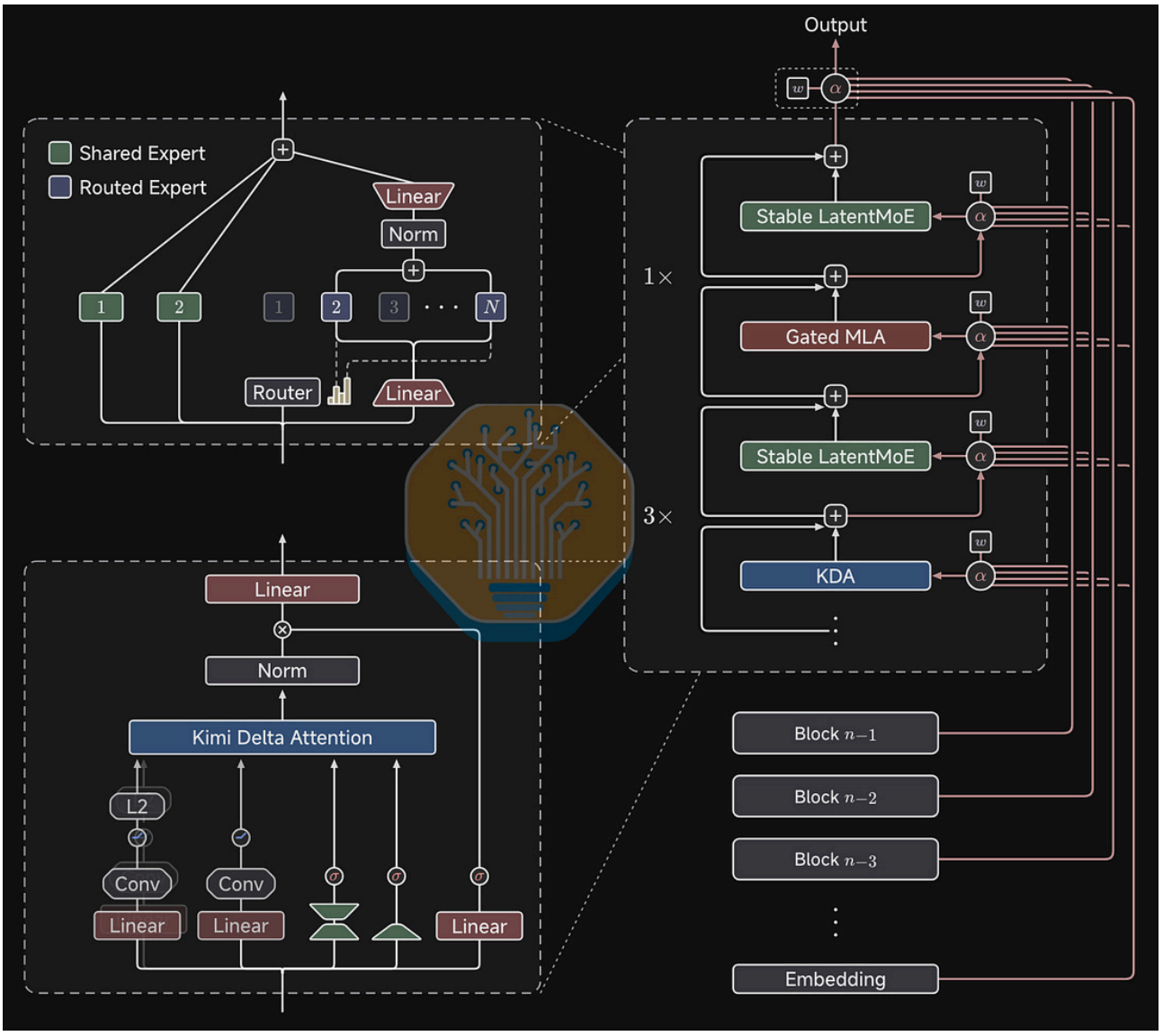
Moonshot trained Kimi Linear models as proof of concept for their KDA design, so we can infer Kimi K3's architecture design from Kimi Linear. Comparing the K3 release tech blog with Kimi Linear, we see Kimi K3 shares the shared expert count, the hybrid linear attention ratio, and the general attention module design.

QQ:1413221435

Config	Kimi Linear	Kimi K3
Num Activated Experts	9 (1 shared 8 routed)	18 (2 shared 16 routed)
Num Total Routed Experts	256 (sparsity 32)	896 (sparsity 56)
KDA to MLA Ratio	3:1	3:1
Activation Function	Swish	Sigmoid Tanh Unit (SiTU)
KDA Output Gate Type	Low rank projection	Linear transformation with bias
Full Attention Type	MLA	Gated MLA
MoE Type	MoE	Stable LatentMoE
Token Routing	Sigmoid + bias	Quantile balancing
Optimizer	MuonClip	Per-head Muon

Source: SemiAnalysis

QQ:1413221435



Source: [Kimi K3 Tech Blog]

The diagram above shows the operations performed on the inputs of KDA. For the query, key, and value, we apply linear transformation and short convolution. Applying short convolution effectively capturing local token dependencies, and doing a left padding convolution avoids breaking causality. We additionally apply L2 norm to the query and key to stabilize the eigenvector of the transition and the output matrices. For the decay memory gates, alpha is a low rank projection, and beta is a down projection. The KDA output is normalized per head and controlled by an output forget gate, implemented as a linear transformation in K3, instead of a low rank projection in Kimi Linear. Finally, we apply a linear layer to mix per-head information.

Kimi Linear interleaves KDA with full attention Multi-head Latent Attention (MLA). Kimi Linear showed that 3:1 is the ideal KDA to MLA ratio that balances performance and efficiency. KDA also serves as a strong position-aware operator, replacing the RoPE in MLA.

QQ:1413221435

Keeping MLA as full attention is an interesting choice, as all other open weight models move to Grouped Query Attention (GQA). MLA uses an absorption trick to reduce the computation of a decode step at the cost of extra computation during the prefill step. This is a sensible trade-off for decode-dominant reasoning workloads, but for prefill-dominant agentic workloads, extra computation becomes a high cost with little benefits. As a result, all frontier open weight models use GQA-based attention mechanisms: GLM 5.2 DeepSeek Sparse Attention, DeepSeek V4 Compressed Sparse Attention, MiniMax M3 MiniMax Sparse Attention, and MiMo V3 HySparse are all based on GQA. We suspect Moonshot's future models such as Kimi K4 will feature attention mechanisms that replace MLA.

SemiAnalysis is a reader-supported publication. To receive new posts and support my work, consider becoming a free or paid subscriber.

✓ Subscribed

QQ:1413221435

KV Cache Efficiency

We argue that **one should not infer KV cache efficiency solely based on KV cache space complexity**. KV cache size is not a standalone factor but a property of the model design: no open weight models are released with static KV cache compression techniques, and model architecture inference efficiency affects KV cache efficiency. The effects of KV cache size also vary, depending on the total memory capacity of a deployed model instance. For example, deploying a model with wide expert parallelism has very different memory profiles than doing so with tensor parallelism, which affects the memory capacity left for KV cache. Thus, we propose considering both the model architecture system efficiency and the KV cache size to understand the KV cache efficiency, and we quantify that with **KV throughput**.

KV Throughput

KV throughput is defined as KV cache size divided by the prefill time (Time to first token), given a specific sequence length. KV throughput represents the minimum bandwidth required to reliably serve a model with PD disaggregation, but it is also a good proxy for understanding KV cache efficiency. Prefill time encapsulates the efficiency of the model architecture, and as the sequence length increases, we will see the memory-bounded and the compute-bounded situations. As shown in the table below, we can see the benefits of hybrid linear attention become more pronounced as sequence length increases.

	Hybrid				Dense	
Seq Len	Kimi Linear	MiMo-V2-Flash	Qwen3.5-397B	Ring-2.5-1T	MiniMax-M2.5	Qwen3-235B
1K	1.19	0.82	4.13	7.27	4.94	4.12
8K	2.29	2.85	6.28	4.47	32.87	22.42
32K	3.87	4.66	8.25	2.59	59.93	33.35
128K	4.88	4.71	7.47	1.46	47.82	21.50

Source: [Prefill-as-a-Service: KVCache of Next-Generation Models Could Go Cross-Datacenter]

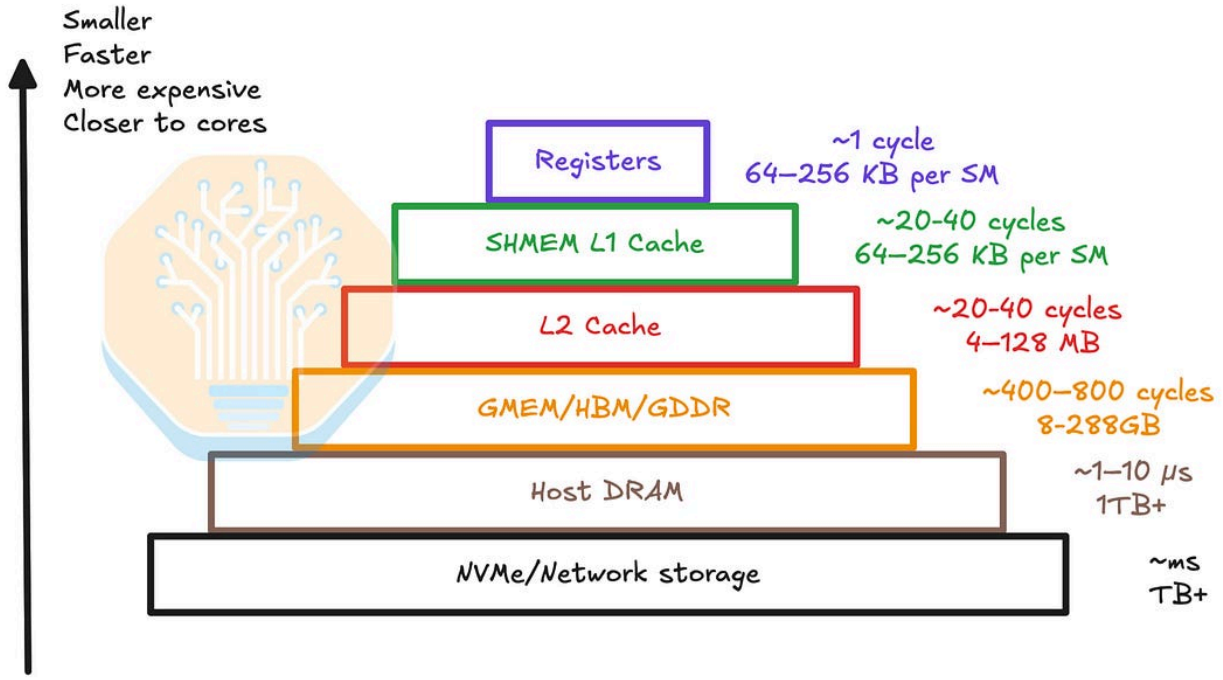
This is also a good way to understand the bandwidth requirements of KV cache offloading to different memory tiers in a cluster.

KV Cache Residency

The location where KV cache is stored follows the memory hierarchy. First, KV cache resides in HBM, the fastest memory in a GPU cluster, consuming whatever capacity is left by model weights and activations. As KV cache size exceeds the HBM capacity, it spills into server DRAM, a higher capacity but lower bandwidth memory pool. Finally, when KV cache exceeds DRAM capacity, it spills to disk storage such as SSD. This is analogous to the computer architecture cache hierarchy: register, cache memory, main memory, disk storage.

The analogy continues for memory coherency. Popular distributed KV cache framework Mooncake Store supports write-through and write-back policies for KV cache loading. Mooncake Store features a distributed KV cache pool that makes all KV cache visible to all workers. Implementing write-through policy between DRAM and the lower-level distributed KV cache pool offers multiple benefits in multi-node scenarios, including sharing prefix cache across nodes, avoid KV cache duplication for tensor parallelized MLA, and KV cache redundancy when a node goes down.

QQ:1413221435
QQ:1413221435



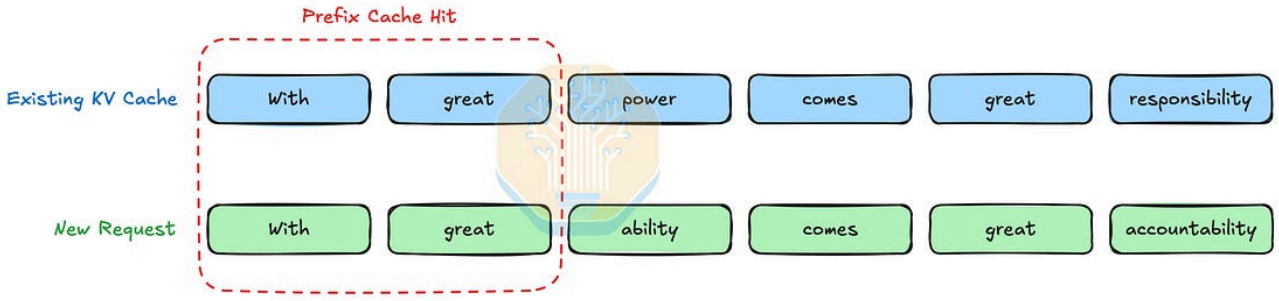
Source: SemiAnalysis

QQ:1413221435

KDA Prefix Cache Management

At each token position in a request, Kimi K3 KDA's recurrent state is fixed in size, whereas standard attention KV cache grows with sequence length. This KV cache space reduction comes at the cost of complicating prefix caching, especially when Kimi K3 is a hybrid attention of KDA and MLA.

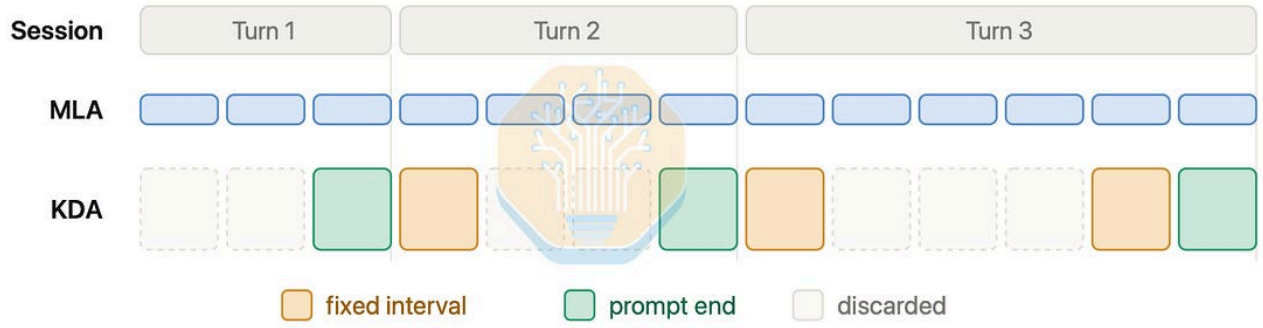
Roughly speaking, modern inference engines identify prefix cache hits by matching the longest token prefix in the existing cache.



Source: SemiAnalysis

Identifying the longest prefix becomes a problem for linear attentions like KDA. Without prior knowledge of where the boundary of a prefix is, we will have to cache KDA's recurrent state at every token position. This means every token has a cache, and the KV cache memory usage regresses to growing with sequence length, defeating the purpose of using linear attention. To tackle this problem, Moonshot saves recurrent states at a coarse granularity, e.g. vLLM caches every 32K tokens. vLLM additionally caches at

prompt boundaries, since for agentic workloads, a new turn typically starts at the end of a prompt.



Source: [Kimi K3 Is Here: Efficient Day-0 Support on vLLM]

This shows that even though linear attentions like KDA greatly reduce KV cache memory consumption, **realistically during serving, they do not consume a constant amount of KV cache memory**.

QQ:1413221435

Attention Residuals

Residual Connections

Residual connections are one of the key innovations that allowed us to build bigger deep neural networks through scaling model depth. The deeper the neural network, the more expressive they become but training them naively is hard. Signals from the earlier layer need to be preserved till the last layer and gradient need to survive from output to first without vanishing.

Instead of modeling whole networks as a single function, passing information only through nonlinear transformations, residual networks connect smaller blocks with identity paths. Each block f_i learns a change to its input x_i, given by the recurrence:

$$x_{l+1} = x_l + f_l(x_l)$$

The identity mapping allows features to carry from shallower units to any deeper unit and gives the gradient a path highway so they do not vanish.

$$\frac{\partial x_{l+1}}{\partial x_l} = I + f_l'(x_l)$$

QQ:1413221435

While residual connections allow us to build deeper networks, they come with challenges.

Early layers heavily influence residual stream to have effect on final output. Because of which residual stream has irreversible information loss with increasing depth. Later layers increase output gain to have effect on this modified residual stream which can destabilize training. Another variant like highway networks allow gating mechanisms for information flow but they suffer from the same crucial problem. Layers don't have selective access to information from earlier layers.

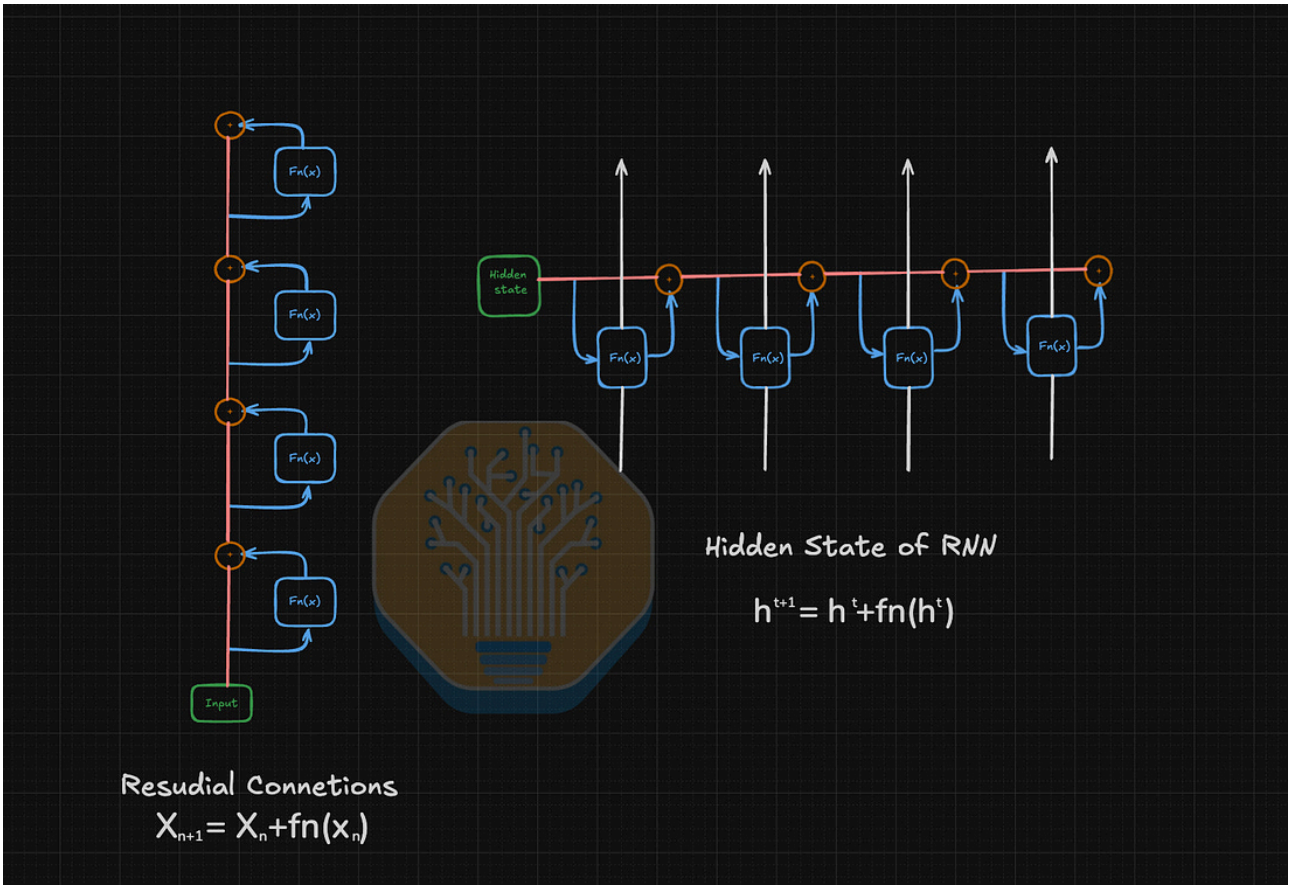
Recurrence In Time and Depth

Sequence modeling dominated by recurrent neural networks had the same recurrence formulation.

$$h_{t+1} = h_t + f(h_t, x_{t+1})$$

Where each step has identity mapping with previous state for direct information flow and the sequence model faced the same challenge: depth in the time axis dilutes signal.

QQ:1413221435



Source: SemiAnalysis

Attention machines transformer removed this constrained by retrieving any token in past with powerful and expensive attention mechanism

QQ:1413221435

Attention on residual stream

Motivated by attention mechanism in sequence modeling, kimi developed attention residual, where they take attention over depth blocks,

$$\alpha_{i\to l} = \frac{\phi(\boldsymbol{q}_l, \boldsymbol{k}_i)}{\sum_{j=0}^{l-1} \phi(\boldsymbol{q}_l, \boldsymbol{k}_j)} \tag{2}$$

For each layer l, we define:

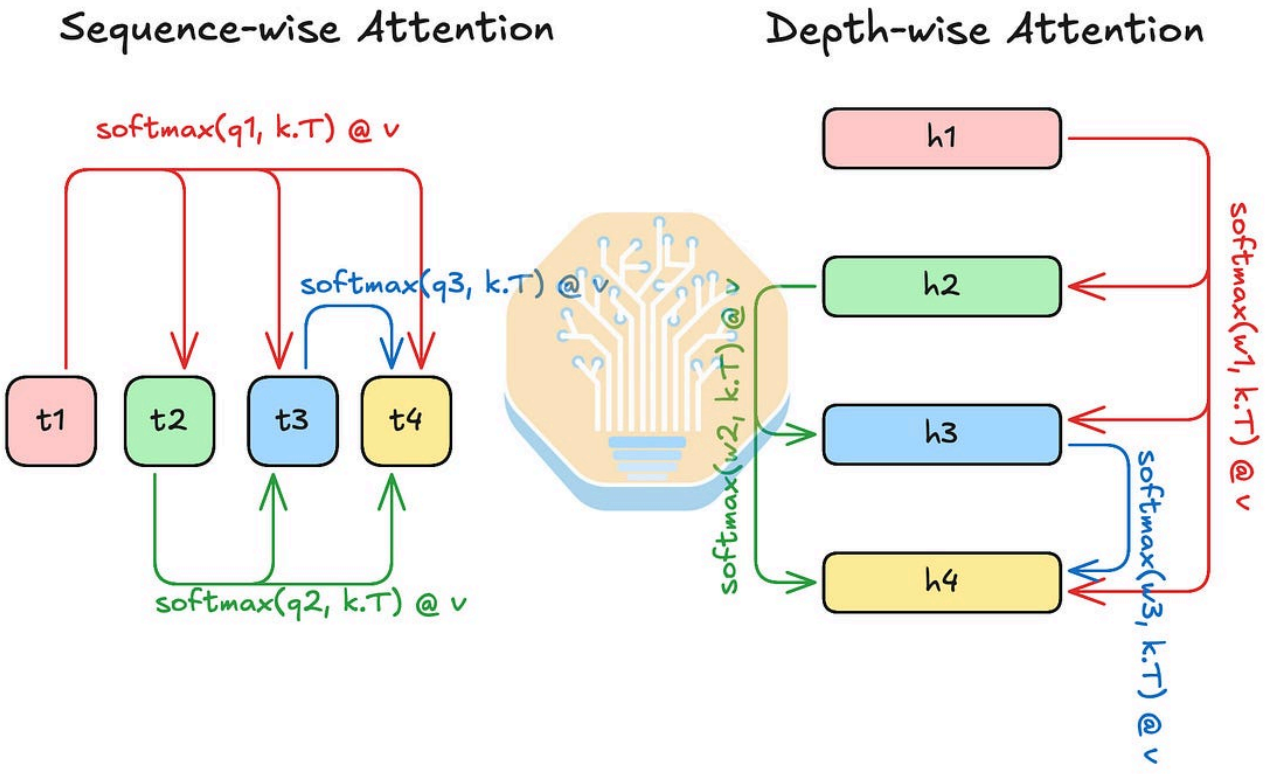
$$\boldsymbol{q}_l = \boldsymbol{w}_l, \qquad \boldsymbol{k}_i = \boldsymbol{v}_i = \begin{cases} \boldsymbol{h}_1 & i = 0 \\ f_i(\boldsymbol{h}_i) & 1 \le i \le l-1 \end{cases} \tag{3}$$

Source: Attention Residuals

Standard causal self-attention computes the output of token t as a weighted sum of previous token representations:

$$\mathbf{o}_t = \sum_{i=1}^{t} \alpha_{i\to t}\, \mathbf{v}_i, \qquad \alpha_{i\to t} = \frac{\phi(\mathbf{q}_t, \mathbf{k}_i)}{\sum_{j=1}^{t} \phi(\mathbf{q}_t, \mathbf{k}_j)}$$

Attention Residuals use the same attention mechanism, but replace the sequence dimension with the depth dimension. Instead of attending over previous tokens, each layer attends over representations produced by previous layers.



Source: SemiAnalysis

Unlike standard attention, the query is a learned parameter for each layer rather than being generated from the current token.

QQ:1413221435

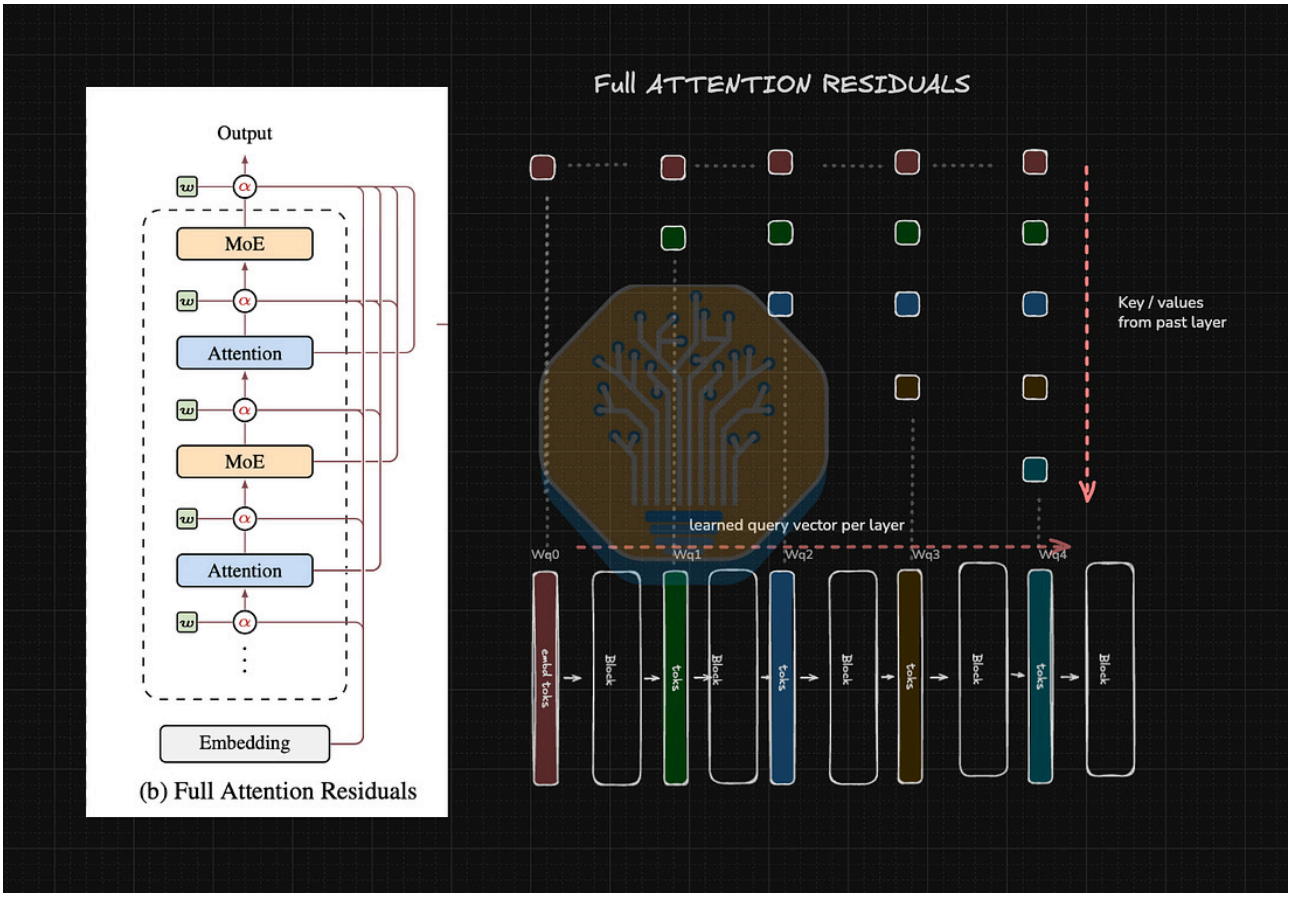
$$\alpha_{i \to l} = \frac{\phi(\mathbf{q}_l, \mathbf{k}_i)}{\sum_{j=0}^{l-1} \phi(\mathbf{q}_l, \mathbf{k}_j)}$$

For each layer ℓ, we define:

$$\mathbf{q}_l = \mathbf{w}_l, \qquad \mathbf{k}_i = \mathbf{v}_i = \begin{cases} \mathbf{h}_1, & i = 0, \\ f_i(\mathbf{h}_i), & 1 \le i < l. \end{cases}$$

Each colored block represents the output token representation from a previous transformer layer. Just as standard causal self-attention performs softmax attention over tokens in the sequence, Attention Residuals perform softmax attention over the representations produced by previous layers.

QQ:1413221435



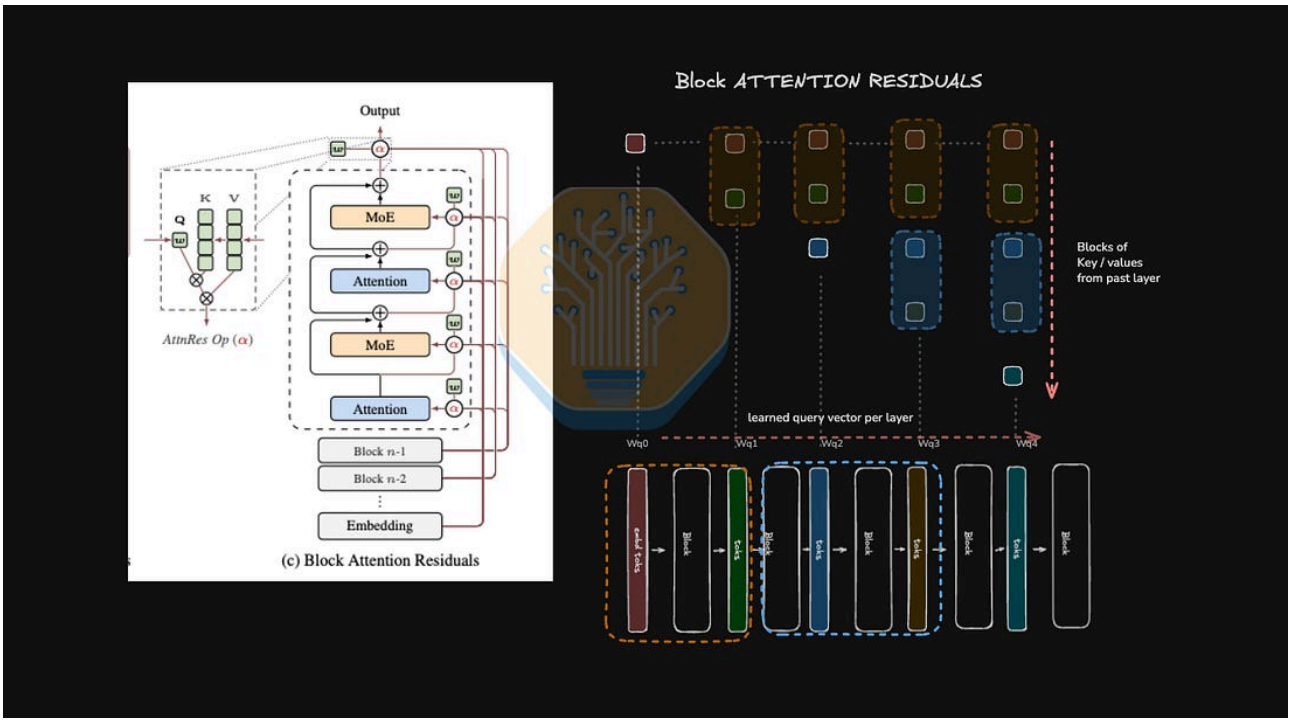
Source: SemiAnalysis

Attention residual allows the model to get fine grained control over what inputs to pick from past layers making the model more expressive.

QQ:1413221435

Block Attention Residuals

Attention residual need to all past layer outputs for attention. For large models distributed over many GPUs this creates O(Ld) communication overhead. To overcome this block attention residual dividends layers L into N blocks of S layers. Block AttnRes applies attention over completed block outputs and for current block its evolving partial sum.



source: SemiAnalysis

Block Attention has minimal tread over full attention residual but they cut down communication from O*(Ld)* to O*(Nd).*

Let b_n^i denote the partial sum over the first *i* layers in block *n*, such that

$$\mathbf{b}_n = \mathbf{b}_n^S, \qquad \mathbf{b}_0 = \mathbf{h}_1.$$

For the *i*-th layer in block n, the available block representations are

$$\mathbf{V}_l = \begin{cases} [\mathbf{b}_0, \mathbf{b}_1, \ldots, \mathbf{b}_{n-1}]^\top, & i = 1, \\ [\mathbf{b}_0, \mathbf{b}_1, \ldots, \mathbf{b}_{n-1}, \mathbf{b}_n^{i-1}]^\top, & i > 1. \end{cases}$$

Unlike standard attention, the query is not input-dependent. Each layer learns a query vector:

$$\mathbf{q}_l = \mathbf{w}_l$$

Attention weights over the available block representations are computed as

QQ:1413221435

The output is the weighted sum of previous layer representations

QQ:1413221435

Rather than depending only on the residual stream to preserve information, Attention Residuals give every layer direct, selective access to earlier representations. This block based variant of attention residuals greatly reduces communication overhead while having competitive performance.

Block residuals show better scaling compared standard residual connection achieving 1.25× compute efficiency. Consistently lower validation loss compared to baseline and gap widening with decay phase. Unlike standard residual networks where output magnitude increases as depth increases. selective aggregation of block attention has bounded output. And consistent gradient magnitude.

✓ Subscribed

Training

Unlike standard residual networks, attention residuals need all N-1 block input for computation of the Nth layer. This becomes a problem for pipeline parallelism as all N layer blocks output need to be transferred across stages.

With clever cross stage caching and activation checkpointing, Kimi reduced overhead to only 4% compared to standard architecture for pipeline parallelism.

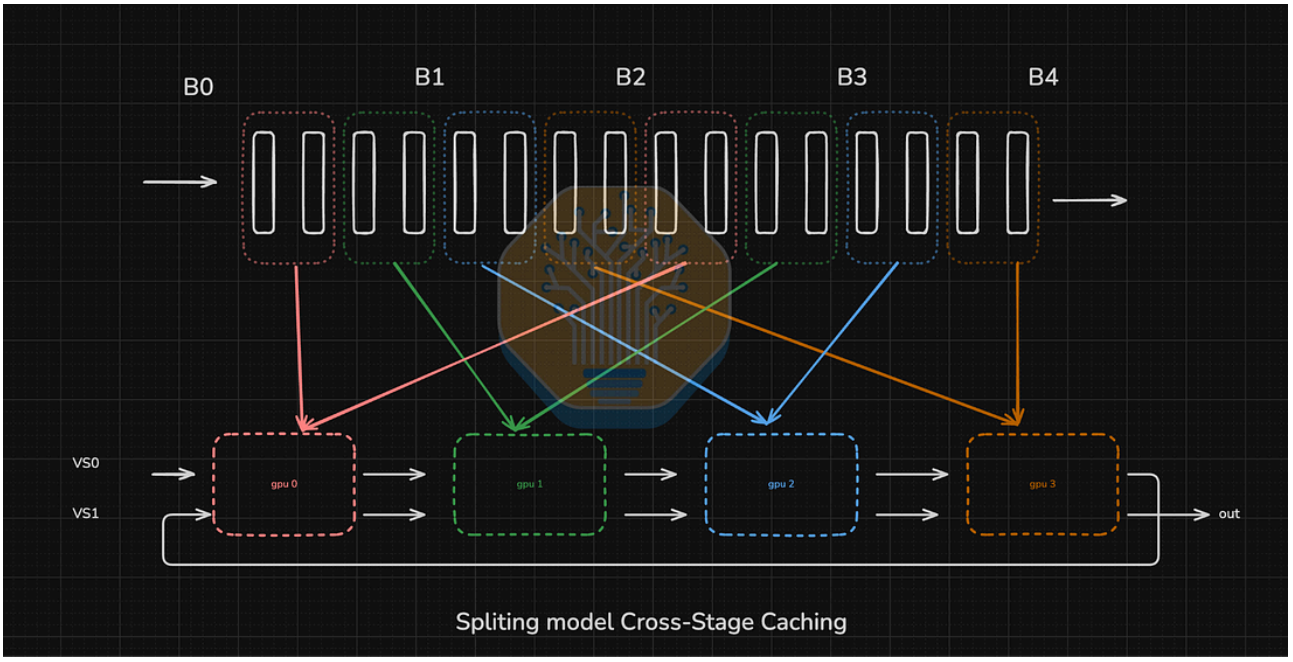
QQ:1413221435

Cross-stage caching

For *P* physical stages and *V* virtual stages. Each block *N* needs *C=PV* communication for each chunk. Naively this needs transferring all accumulated blocks for each stage. This is quadratic cost growth for each physical and virtual stage

$$\text{Comm}_{\text{naive}} = \sum_{j=1}^{C-1} jN_p \cdot d = \frac{C(C-1)}{2} N_p d$$

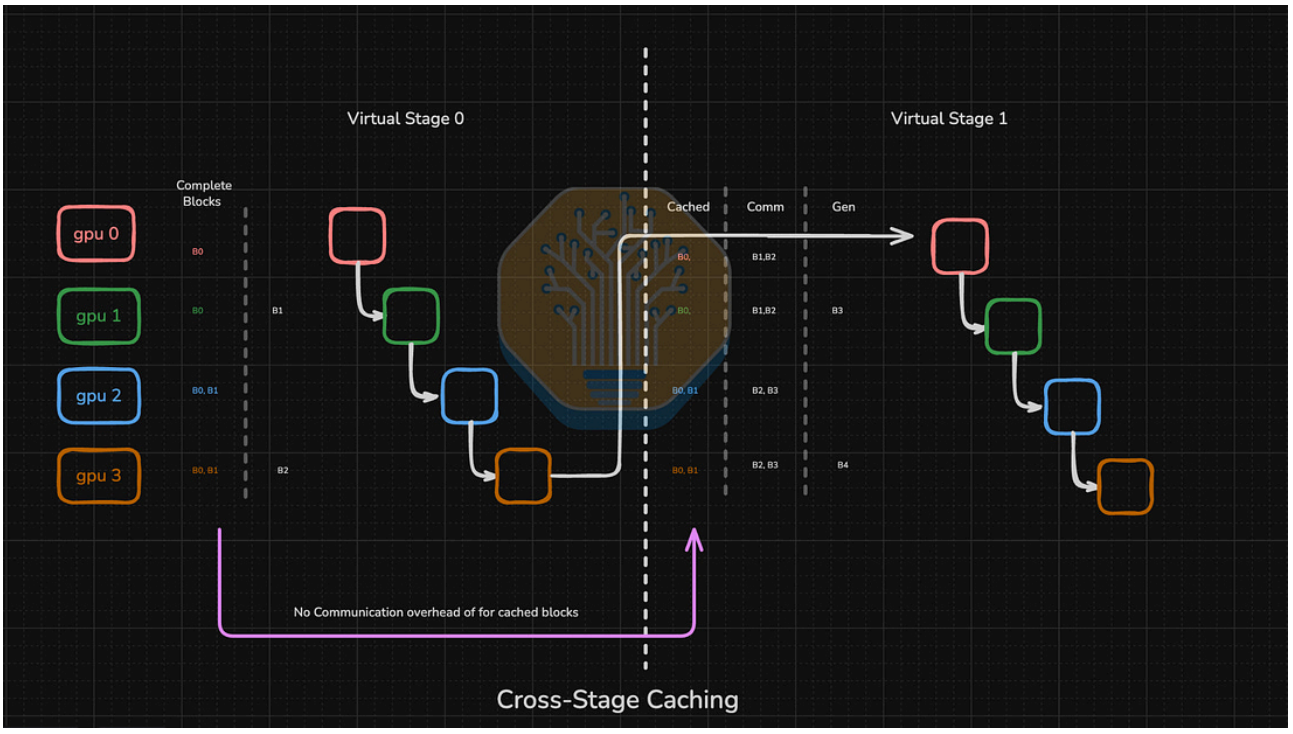
This high communication can be reduced by caching input across virtual stages. Blocks computed in earlier layer can be stored in local memory,



Source: SemiAnalysis

For the first virtual stage all block embedding needs to be transferred in the physical stage, each completed block is stored on respective rank. For all subsequent virtual stages all cached blocks can be reused for computation. Only the block not present on rank need to be transferred for attention to residual computation.

QQ:1413221435
QQ:1413221435



Source: SemiAnalysis

QQ:1413221435

These split communication costs for first and subsequent virtual stages. For the first virtual stage its need incur the same quadratic cost for all physical layers. In subsequent virtual stages we need cached inputs from local devices and Transfer of only PNp chunks needed cutting down total communication from O*(C)* to O*(P)*

$$\text{Comm}_{\text{cached}} = \underbrace{\frac{P(P-1)}{2} N_p d}_{\text{first virtual stage}} + \underbrace{(V-1)P^2 N_p d}_{\text{subsequent virtual stages}}$$

The cutdown of communication is directly proportional to virtual stages V. Because of this for full stage of one forward and backward pass all computation and communication can be overlapped

Memory overhead

Due to cross stage caching all blocks are stored once across all V virtual stages. With Activation checkpointing all inter-block chunks for attention are eliminated. Each stage activation checkpoint *Pl* matches memory size of H*l* of standard architecture and has no extra memory cost.

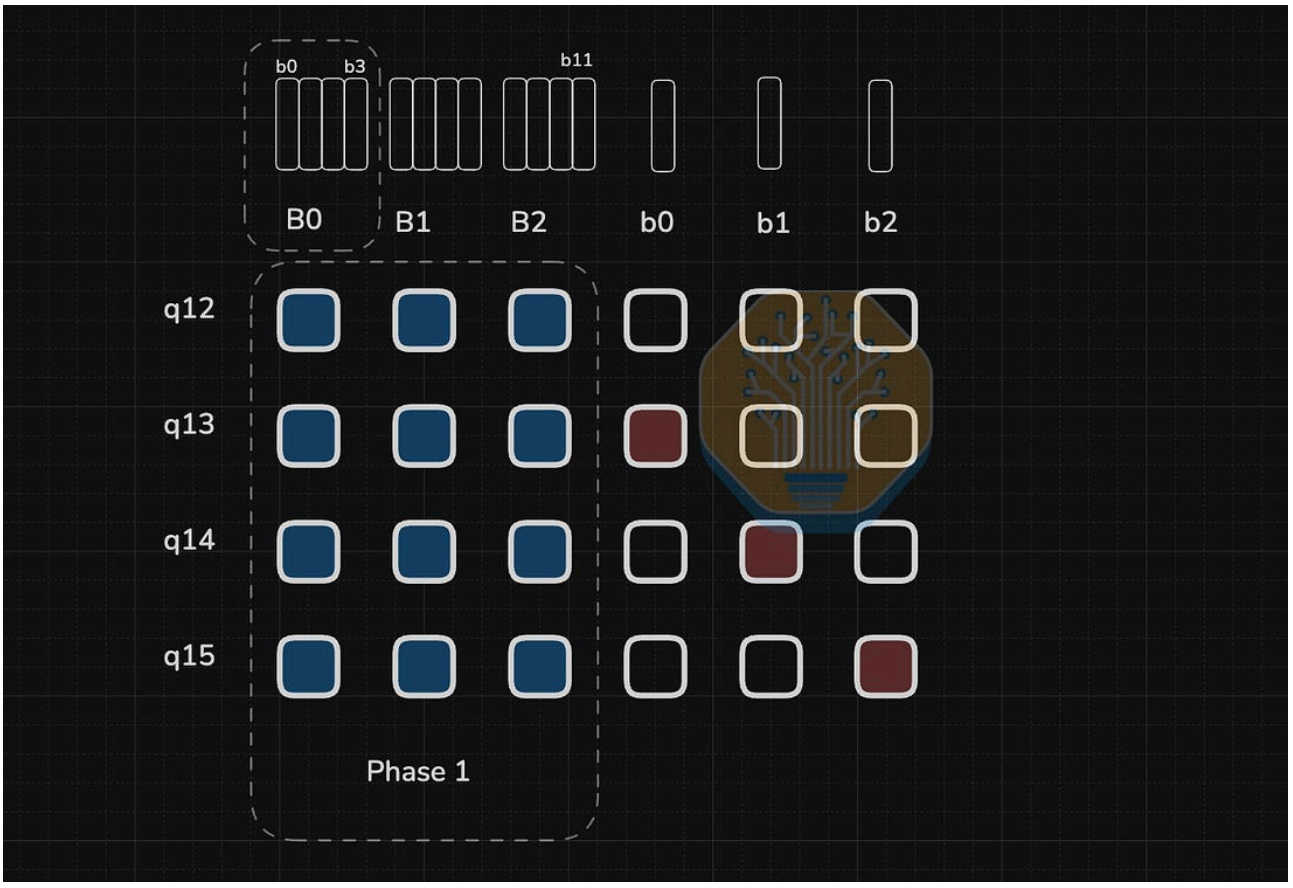
Inference

Because Attention Residuals need the output of all previous blocks to compute attention, a naive implementation has excessive memory accesses. To reduce overhead, inference is split into two phases which mirror prefill and decode stages of autoregressive attention.

QQ:1413221435

This computation is divided into inter-block attention for completed blocks and intra-block attention for evolving attention in the running block.

Phase 1: Parallel Inter-Block Attention

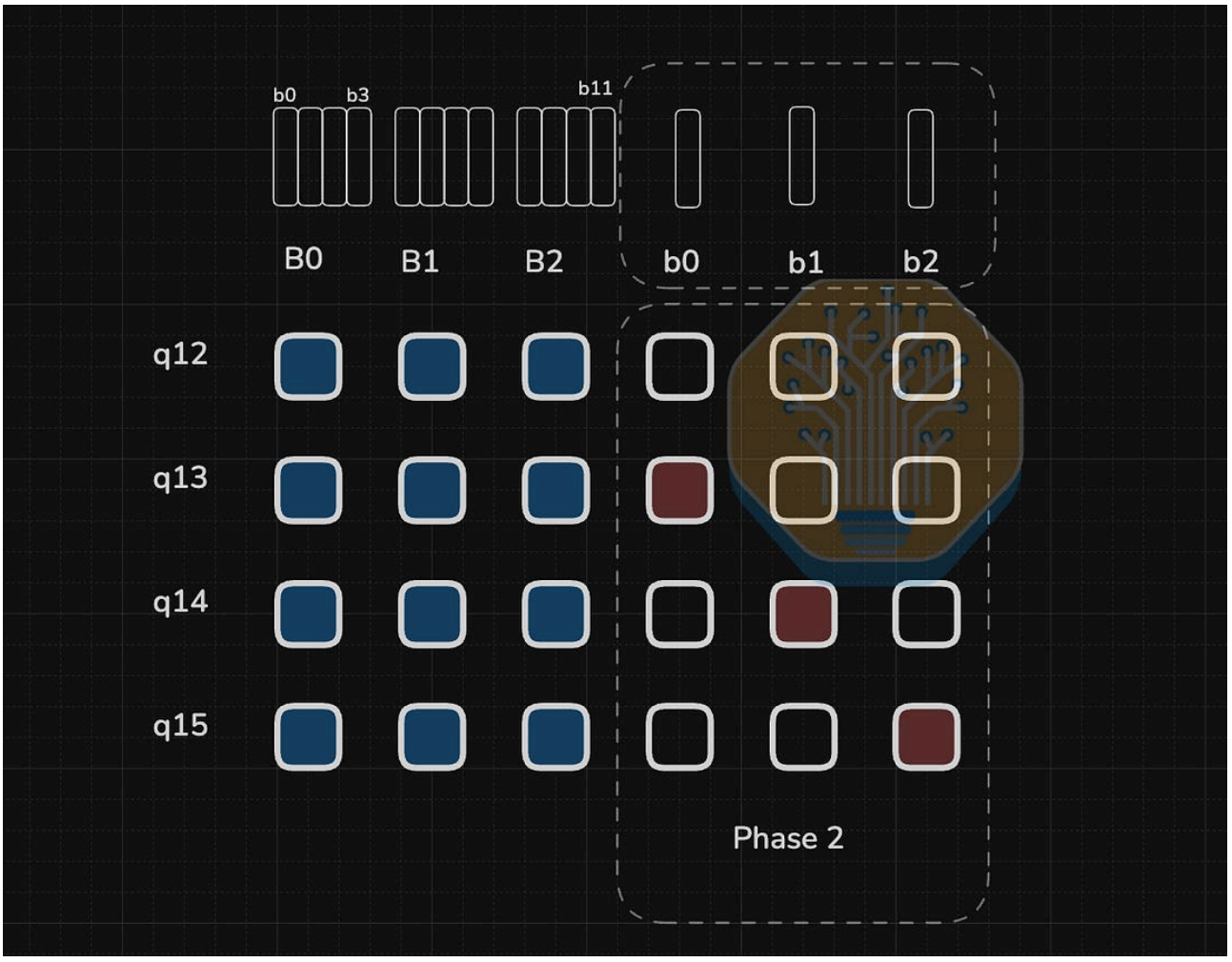


Source: SemiAnalysis

During decoding, we have to output the completed block and the query vector learned per layer. All inter block layers simultaneously with a single batched query against the completed block representations, returning both outputs and softmax statistics which can be reused for further computation. This phase is similar to prefill phase decoding

Phase 2: Sequential Intra-Block Attention

QQ:1413221435
QQ:1413221435



Source: SemiAnalysis

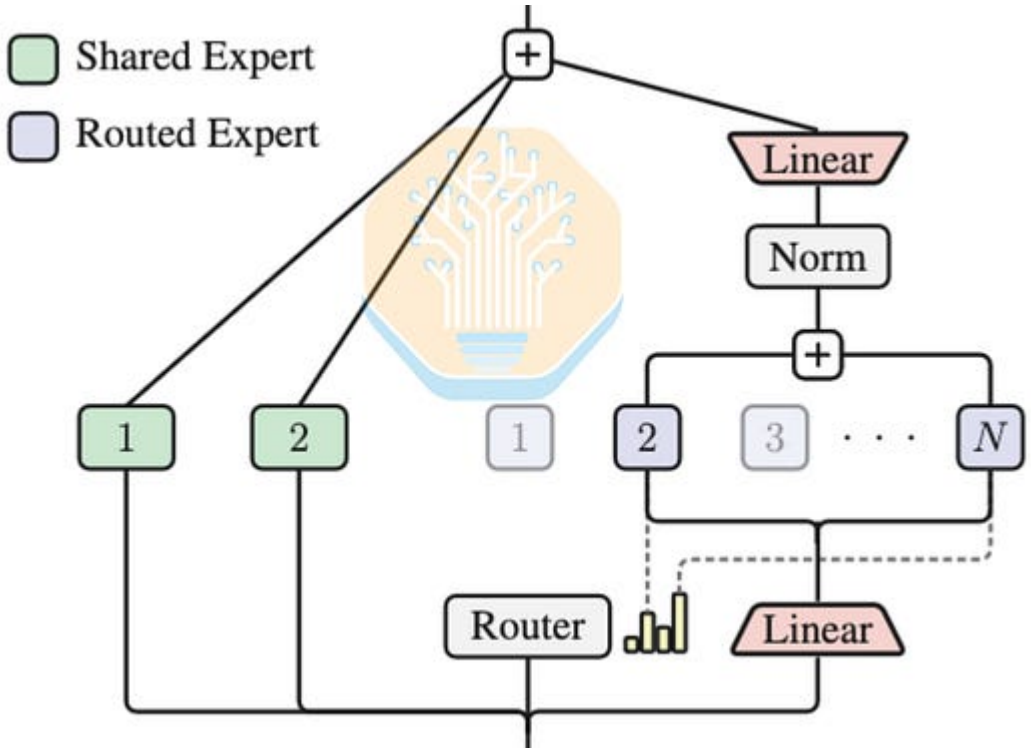
This phase is analogous to the decode phase, Similar to flash attention, evolving sum can be computed with online softmax for intra blocks combined with precomputed inter block results. Which reduces redundant memory access.

With this two phase design, the IO footprint is similar to standard residual architecture, with only the addition of phase ones inter block computation, amortized by batching all queries in the block.

LatentMoE

LatentMoE compresses the routed tokens before the dispatch operation, and then decompresses them after the aggregation operation. In Kimi K3’s Stable LatentMoE, they apply an RMSNorm before the up-projection (decompressing) operation to reduce sensitivity to scale variations and improve model performance.

QQ:1413221435
QQ:1413221435



Source: Kimi K3 Tech Report

Here we explain the design principles behind LatentMoE regarding MoE communication. As shown in the LatentMoE paper, the communication volume is proportional to total routed tokens t, number of active experts K, and expert input dimension d, while being inversely proportional to the expert parallel size E. This is potentially the reason behind Kimi K3's latent MoE dimension size and active expert count configuration. Kimi K2 series feature 8 active experts with input dimension size 7168, so Kimi K3's latent input dimension size being 3584 (half of 7168) would allow the active expert count to double to 16 without increasing the communication volume.

However, the **ratio of communication to computation time** is arguably more important for estimating system efficiency (Discussions here and here). The ratio indicates the roofline of how well MoE kernels can overlap communication with computation at a throughput-bound regime, and **expert intermediate dimension size** is the only model configuration that affects the ratio. Specifically, increasing the expert intermediate dimension size would decrease the ratio, meaning that the theoretical maximum fraction of communication that can be hidden is higher. Here we derive the formula:

- t: total input tokens across the expert parallel (EP) domain
- K: number of active experts per token
- N: number of total experts
- E: Ranks in the EP domain
- d: Expert input dimension

QQ:1413221435

- *m*: Expert intermediate dimension
- *P*: Aggregate bytes communicated per activation element (dispatch + combine)
- *F*: Effective FFN expert (modeled as SwiGLU) computation throughput per GPU, FLOP/s
- *B*: Effective uni-directional network bandwidth per GPU, B/s

1. Assuming uniform expert routing, each GPU is assigned *t * K / E* tokens
2. Assuming uniform expert routing, an average *1 / E* tokens are local to the source GPUs, so each GPU dispatches *(t*K/E) * (1-1/E)* tokens
3. Each token is a d dimensional vector, so the communication volume per token is *d * P*
4. The communication volume per GPU is *(t*K/E) * (1-1/E) * d * P*
5. The communication time T_comm = *(t * K * d * P) / (E * B) * (1-1/E)*
6. The SwiGLU computation involves 3 matrix multiplications:
    a. Up (First) projection: *d* to *m*
    b. Gate projection: *d* to *m*
    c. Down (Second) projection: *m* to *d*

QQ:1413221435

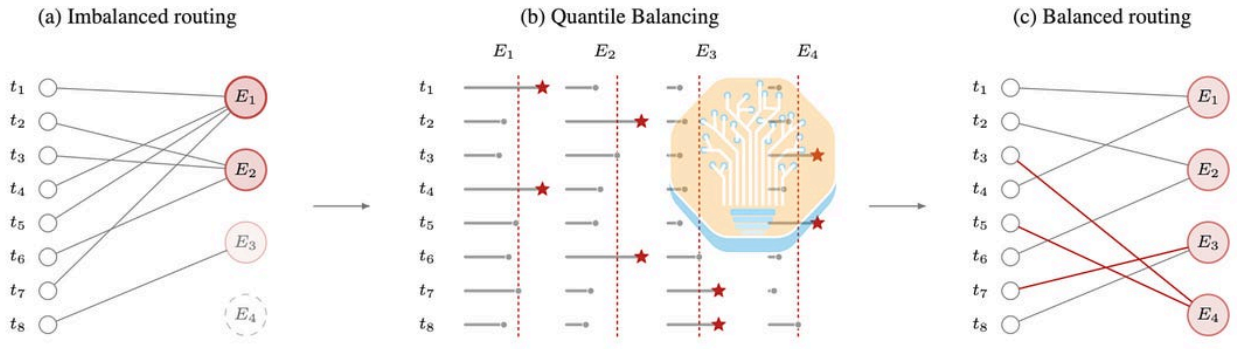
So the computation is *2*d*m + 2*d*m + 2*m*d = 6*d*m* FLOPs per token

7. The computation time per GPU is *T_comp = (6*d*m) * (t*K/E) / F*
8. The communication to computation time ratio is
   T_comm / T_comp
   *= ((t * K * d * P) / (E * B) * (1-1/E)) / ((6*d*m) * (t*K/E) / F)*
   *= (P*F) / (6*m*B) * (1-1/E)*

We believe this formula also motivates an increase in expert intermediate dimension to 3072 in not just Kimi K2 to K3, but all recent open weight models, including DeepSeek V4 Pro, MiniMax M3, MiMo V2.5 Pro, and Inkling. As hardware improves and expert weight precision reduces to save memory capacity, the compute throughput increases, so one way of reducing the ratio is by increasing the expert intermediate dimension.

Quantile load balancing (QB)

QQ:1413221435



Source: Kimi K3 report

Many previous load balancing methods require careful hyperparameter tuning. Quantile balancing is hyperparameter free aux-loss free load balancing technique developed by Jianlin Su in Feb 2026 blog post

Base principle QB is the same as auxfree load balancing where router biases are updated dynamically based on the system's load. But instead of updated bias by some small coefficient like aux-free lb, QB directly computes the next bias from the distribution of router scores relative to routing cutoff threshold. Bias updates become small naturally when the router balances load evenly.

QQ:1413221435

Algorithm 1: The alternating QB solver.

Input: score matrix $\boldsymbol{s} \in \mathbb{R}^{m \times n}$
Output: assignment $\boldsymbol{x} \in \{0,1\}^{m \times n}$

Initialize $\boldsymbol{\beta} = \mathbf{0}_{1 \times n}$;
for $t = 1, 2, \cdots, T$ **do**
  $\boldsymbol{\alpha} \leftarrow \text{desc_sort}(\boldsymbol{s} - \boldsymbol{\beta},\ \text{axis=1})_{[:,\, k:k+1]}$
  $\boldsymbol{\beta} \leftarrow \text{desc_sort}(\boldsymbol{s} - \boldsymbol{\alpha},\ \text{axis=0})_{[mk/n:\, mk/n+1]}$
end
return $\boldsymbol{x}$ *with* $x_{i,j} = 1$ if $j \in \text{argtop}_k(\boldsymbol{s}_i - \boldsymbol{\beta})$, and 0 otherwise

Source: Kimi K3 report

QB tries to find the bias that would have approximately balanced under the current cutoffs and routing on the current batch, solving constraint optimization problems and applying these updates for the next batch. The first constraint is that each token is routed to exactly k experts. The second constraint is a batch of m tokens each picks k experts, gives (mk) assignments in total, to spread load evenly across n experts each expert should process $q=mk/n$ tokens.

Each token finds the cutoff threshold as the (k+1)-th highest biased router score and uses it to calculate the bias update needed to balance load for each expert. For each expert, QB sorts the margins between its router score and every token's cutoff. It sets negative

QQ:1413221435

bias to q+1 the largest margin, leaving exactly q margin above the threshold. Since q/m=k/n, this is (1-k/n) quantile of the margin, which is why it's called Quantile Balancing.

SemiAnalysis is a reader-supported publication. To receive new posts and support my work, consider becoming a free or paid subscriber.

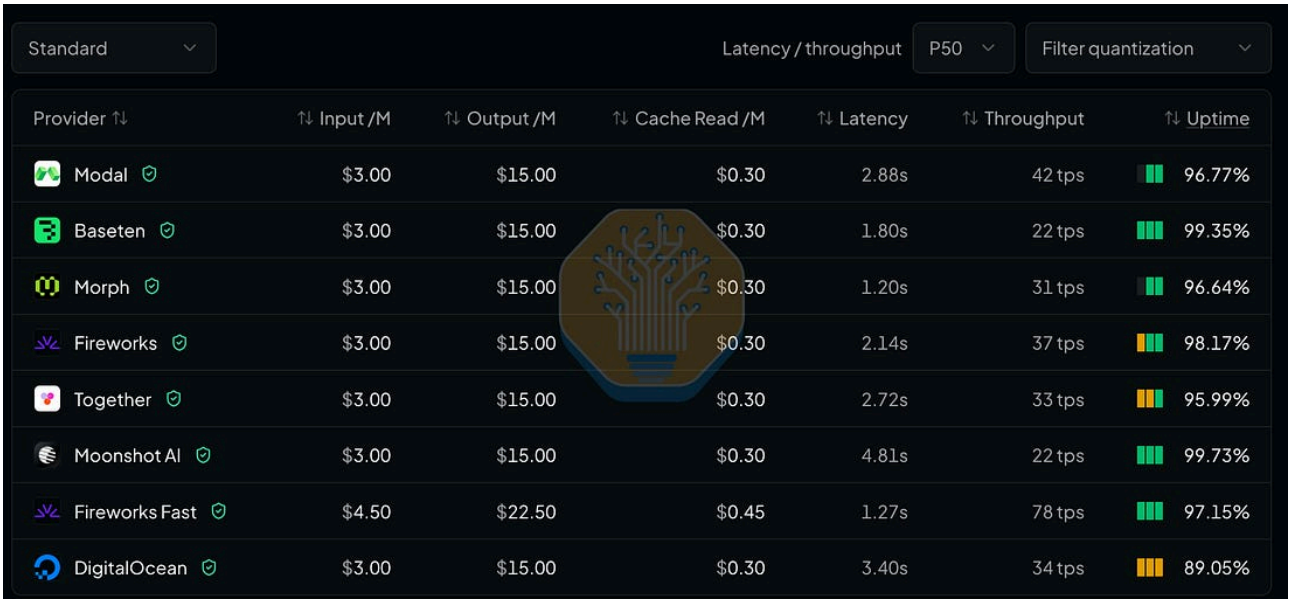
✓ Subscribed

Inference performance

We are actively tracking Kimi K3's inference performance on InferenceX.

As of 30th July, all providers on OpenRouter have a floor of $3 per million tokens input and $15 per million tokens output. Both Nvidia and AMD had Day 0 recipes on vLLM, boasting DRAM offload and DSpark speculative decoding.

QQ:1413221435



Provider	Input /M	Output /M	Cache Read /M	Latency	Throughput	Uptime
Modal	$3.00	$15.00	$0.30	2.88s	42 tps	96.77%
Baseten	$3.00	$15.00	$0.30	1.80s	22 tps	99.35%
Morph	$3.00	$15.00	$0.30	1.20s	31 tps	96.64%
Fireworks	$3.00	$15.00	$0.30	2.14s	37 tps	98.17%
Together	$3.00	$15.00	$0.30	2.72s	33 tps	95.99%
Moonshot AI	$3.00	$15.00	$0.30	4.81s	22 tps	99.73%
Fireworks Fast	$4.50	$22.50	$0.45	1.27s	78 tps	97.15%
DigitalOcean	$3.00	$15.00	$0.30	3.40s	34 tps	89.05%

Source: OpenRouter

On InferenceX, we benchmark Kimi K3 serving performance directly on recorded internal claude code traces. We replay an hour of these traces as they reach a steady state. There is a median of 142k input tokens and a median of 444 output tokens per turn with a median of 65 turns per session. The short output tokens per turn is typical for workloads on agentic harnesses, where the agent calls tools frequently, even edits are tool uses.

QQ:1413221435

This benchmark is a big step up from our previous 8k1k/1k1k benchmark, as it truly reflects real-world agentic use cases. From a systems perspective, it is also realistic and

closest to production systems. It can reflect KV cache behavior, including prefix cache and KV offloading to DRAM.



Source: InferenceX

For Kimi K3, Day 0 bringup was easier than DSv4 due to better documentation and preparation ahead of weights release. Appropriate images and a speculative decoder model were released at the same time as the weights.

DeepSeekV4 1.6T Day 0 to Day 43 Performance Over Time - Huawei, GB300 NVL72, MI355X, B200

BRYAN SHAN, CAM QUILICI, AND 5 OTHERS·JUN 9

QQ:1413221435
QQ:1413221435

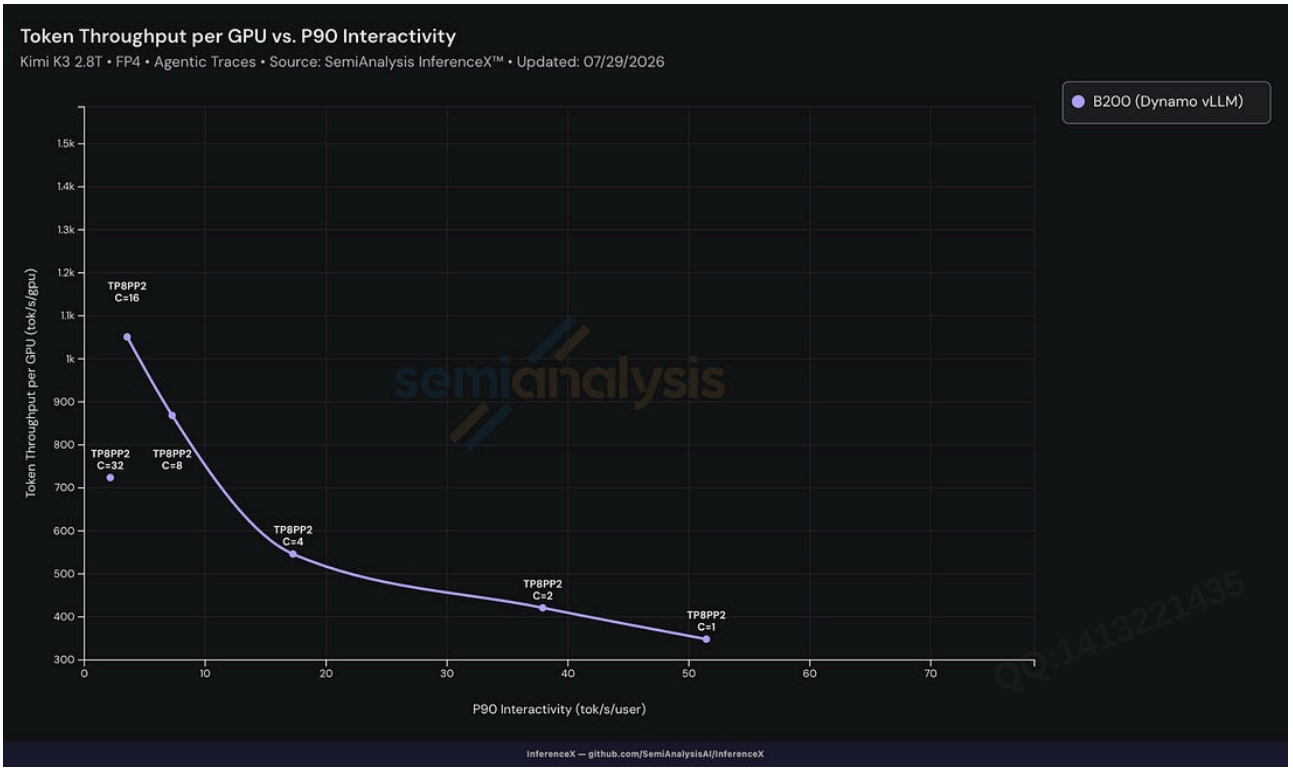


QQ:1413221435

The release of DeepSeek v4 marks another step forward for the open model community - unsurprisingly, it is the product of a Chinese lab. The evolution of its performance over time is of paramount importance to the AI Ecosystem. The open-source InferenceX engineering team has pulled multiple all-nighters to measure performance results for this model on Day 0, Day 1, Day 2, and beyond and bring these results to the world.

Read full story

For Nvidia, bringup was simple. But due to the models' sheer size, it doesn't fit on a single B200 node. We had to use PP to get it working. DSpark also didn't work with PP.

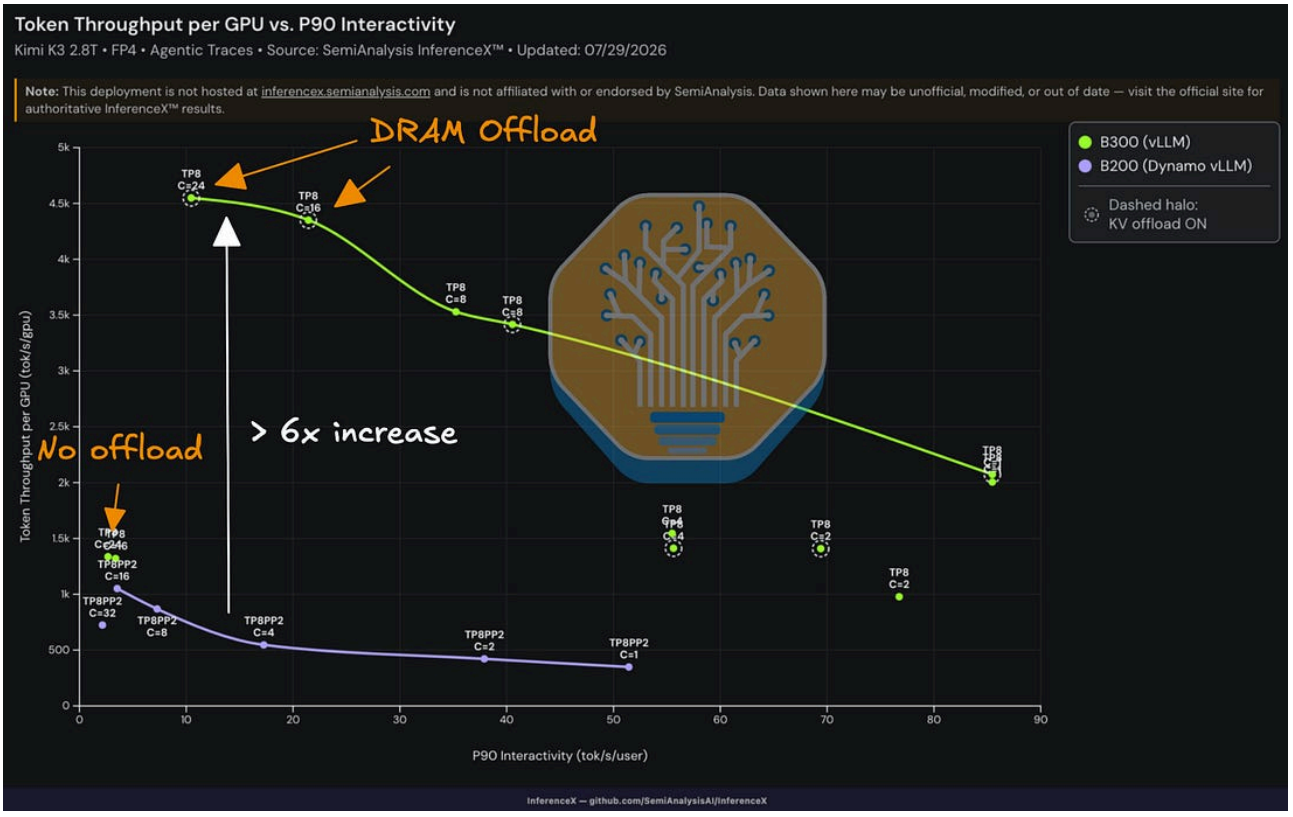


Source: InferenceX

For B300, the model fits on 1 node and serves well. After accounting for the weights, GPU HBM can only hold 3.25M tok. In the graph below, throughput goes up as batch sizes increase until concurrency increases above 8. This roughly correlates to the 3.25M tok KV cache budget, and cache starts to thrash, resulting in hit rates falling to < 10% when theoretical hit rate is 95%.

But for a system with 219.91 GB/rank CPU DRAM offload, this results in an additional 15.76M tokens worth of KV cache, or a 4.85× total tokens worth of KV cache. And the performance gain is huge. With DRAM offload and without speculative decoding, This enables configs with concurrency above 8 to run without cache thrashing.

Cache thrashing configs appear in the bottom left of the throughput-interactivity graph, and are characterized by very low cache hit rates.



Source: InferenceX

TCO

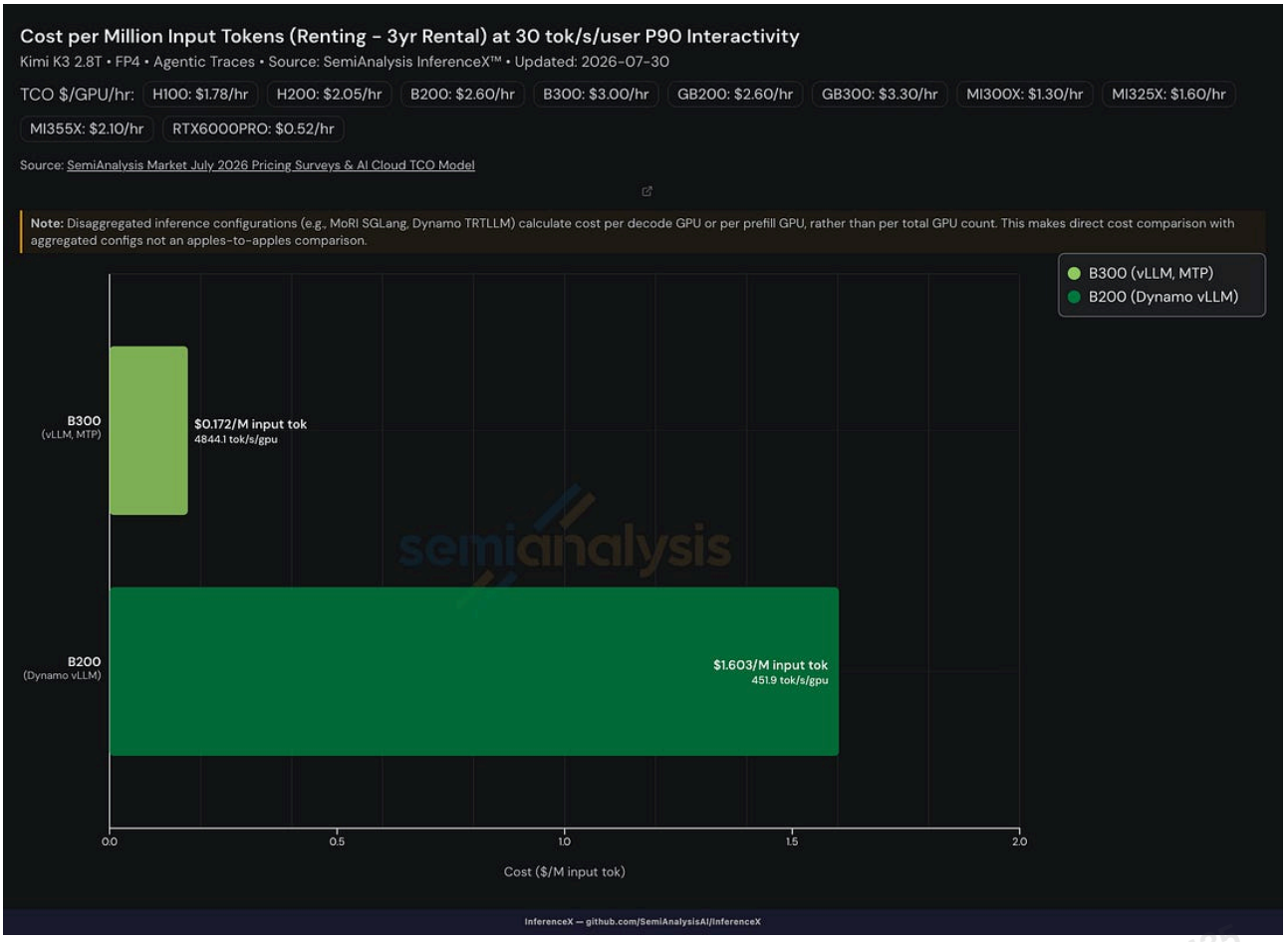
From SemiAnalysis claude code and codex usage, costs from cache read make up most of our spending.

QQ:1413221435

Token cost breakdown	
Cost type	Portion
Input	5.80%
Cache Read	52.70%
Cache Write	27.60%
Output	13.90%

Source: SemiAnalysis

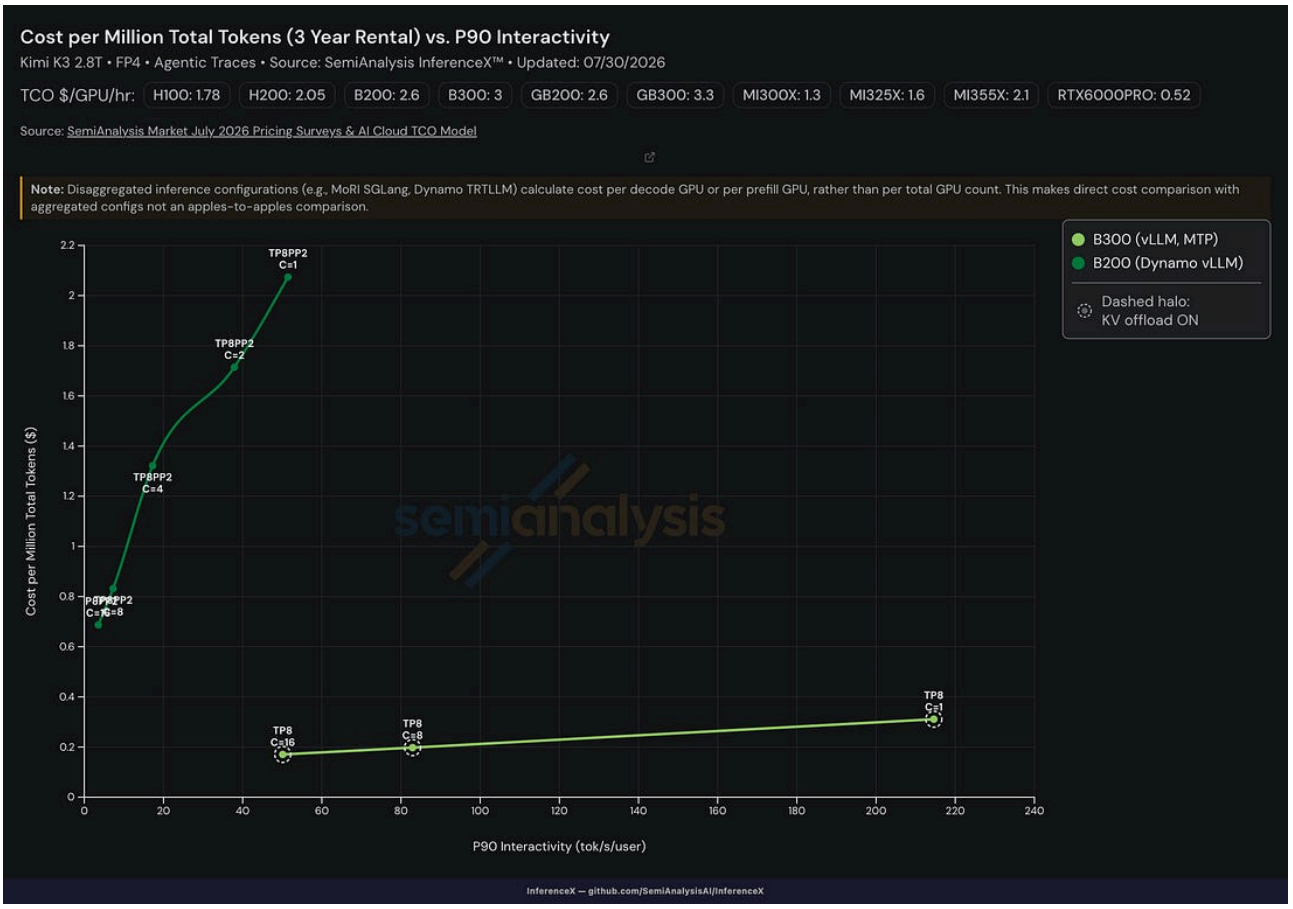
At a P90 of 30 tok/s/user, serving with 1 node of B300 utilizing DSpark speculative decoding, gives you an input throughput of 4844.1 tok/s/GPU, translating to a cost of $0.1712/M input tokens for 3Y rental prices of $2.60/hr as of July 2026. This means providers are offering at an order of magnitude more than cost on serving Kimi K3. On B200 this is much worse and nearly infeasible.



Source: InferenceX

Using our AgentX dataset, which has 315:1 ISL to OSL ratio, we estimate the blended price of using Kimi K3 on Moonshot's platform is $0.74 per million tokens at 22 tok/s/user.

At a similar interactivity, our InferenceX results shows serving Kimi K3 on B300 with vLLM costs $0.171 per million tokens at more than double the interactivity, a 76.89% reduction in cost.



Source: InferenceX

QQ:1413221435
QQ:1413221435