DEEPSEEK INFERENCE THEORETICAL MODEL

Deriving the performance from hardware primitives

Authors: Piotr Mazurek and Eric Schreiber @ Aleph Alpha GmbH, September 2025

DeepSeek Inference Theoretical Model

Deriving the performance from hardware primitives

Piotr Mazurek and Eric Schreiber @ Aleph Alpha GmbH

September 2025

As of August 2025, DeepSeek v3 remains the most popular open-source model, according to OpenRouter (see Fig. 3). The DeepSeek team introduced a series of substantial inference-time optimizations, making the model surprisingly efficient to serve, despite its massive (671B parameters) size. To better understand the impact of the architecture and the applied optimizations, we designed a theoretical model estimating the DSV3/R1 throughput numbers given specific hardware parameters. The model is a "digital twin" that enables the estimation of performance without the need to run the experiments.

Our goal is to provide insight into the tradeoffs between latency, throughput, and cost with respect to different hardware options. While our model follows the DeepSeek V3/R1 architecture, a simple extension makes it possible to apply it to other models that build on the DeepSeek architecture, such as Kimi K2. We demonstrate that (depending on the batch size) considering the number of involved GPUs and speed of interconnect between them, different factors (such as compute, memory, or communication bandwidth) can limit the end-to-end performance. We hope that the model provides useful insights when building intuitions about the topic of inference of large-scale "mixture of experts" (MoE) models.

1 DeepSeek MoE Architecture

DeepSeek V3/R1 are two prominent examples of "mixture of experts" (MoE) models. The key challenge in MoE inference is that, unlike in dense models like Llama3, each processed token activates only a subset of parameters, rather than the entire model. The decode phase is primarily memory bound, meaning that the majority of the execution time, and as a result most of the cost associated with running the model, comes down to the time of loading the model's parameters from the global memory. This property of dense models naturally incentivizes amassing a batch as big as possible and sharing the cost of loading the model parameters across as many requests as possible. This achieves a sort of economy of scale: a fixed cost shared by multiple users.

For MoEs this becomes substantially more challenging. During the decode phase, each token in the batch activates only a small subset of parameters at every layer. This means that each request requires us to load a different part of the model, as demonstrated in Fig. 1. As the number of requests in a batch increases, a more and more substantial portion of the model needs to be loaded from global memory. The experts are chosen semi-stochastically[1], so some of the tokens in the batch will be routed to the same expert. As we progressively increase the batch size, more and more experts will be shared by different requests. This means that at larger batch sizes we partially recreate the situation from the dense model: sharing the cost of model loading between

[1]we elaborate on this in a later part of the text

multiple users. Unfortunately, this means that we need significantly more requests, that is, more users, to achieve the same economy of scale for MoE models.

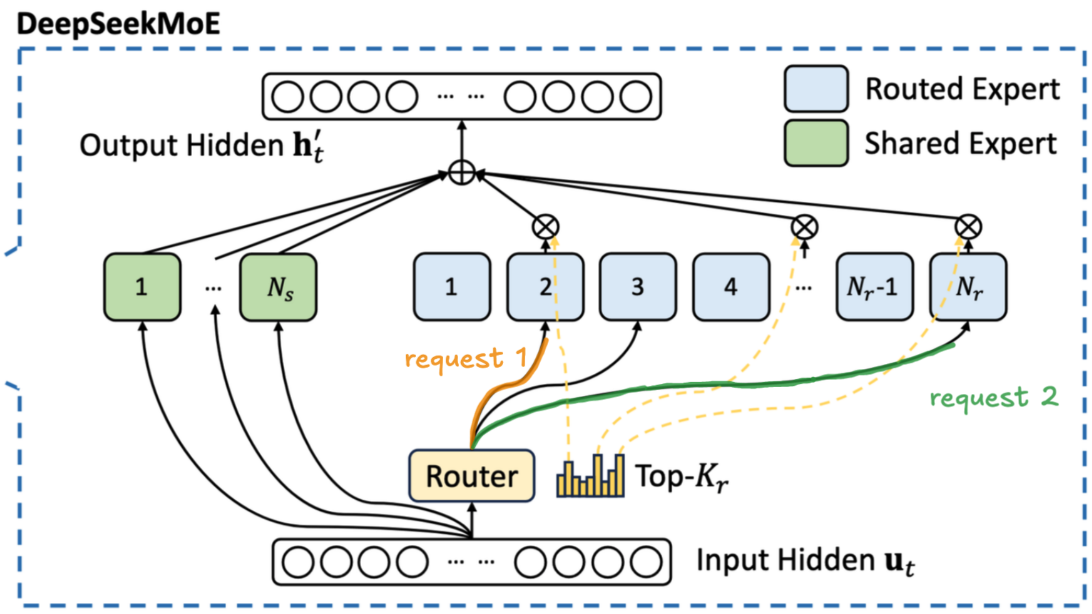


Figure 1: *Two request tokens activating different parts of the model, requiring us to load more weights, saturating the memory bandwidth*

To illustrate this challenge: while DeepSeek V3 activates only 37B parameters for a single forward pass, this number grows nearly linearly with batch size as different queries activate different experts. At large batch sizes, the system may need to load close to the full 671B parameter model, creating severe memory-bandwidth bottlenecks. Furthermore, this need for bigger batches requires substantially more resources to store the key-value (KV) cache for all of the requests. These two factors necessitate running the model beyond a single node. Ultimately, there may not be enough memory bandwidth, and there may not be enough space in memory on a single node to amass enough users to make model serving economically viable.

A GPU node is a specialized computer system designed with high-performance computation in mind. It is essentially a server with multiple GPUs, alongside hardware like CPUs, memory, or networking equipment. A popular example of a node would be a DGX system by NVIDIA, containing eight high-end GPUs (such as H100s, H200s, or B200s) in a single chassis, along with high-speed interconnects between the GPUs (NVLink). Within the node, the GPU-to-GPU (*intra-node*) communication is possible at much higher speeds than the communication between the nodes (*inter-node*).

To effectively host large scale MoE-style models, an inference provider needs multiple GPU nodes. Ideally, we would like to split the model so that each GPU handles some subset of the experts and routes all relevant queries to this GPU. This way each GPU stays busy, and the GPUs do not need to communicate intermediate results as in tensor parallel (TP) setups. This approach is called expert-parallelism (EP). Note that expert choosing occurs at every layer, for every token. During the decode phase, as the model generates token after token, the new token is routed to different experts, located on different GPUs.

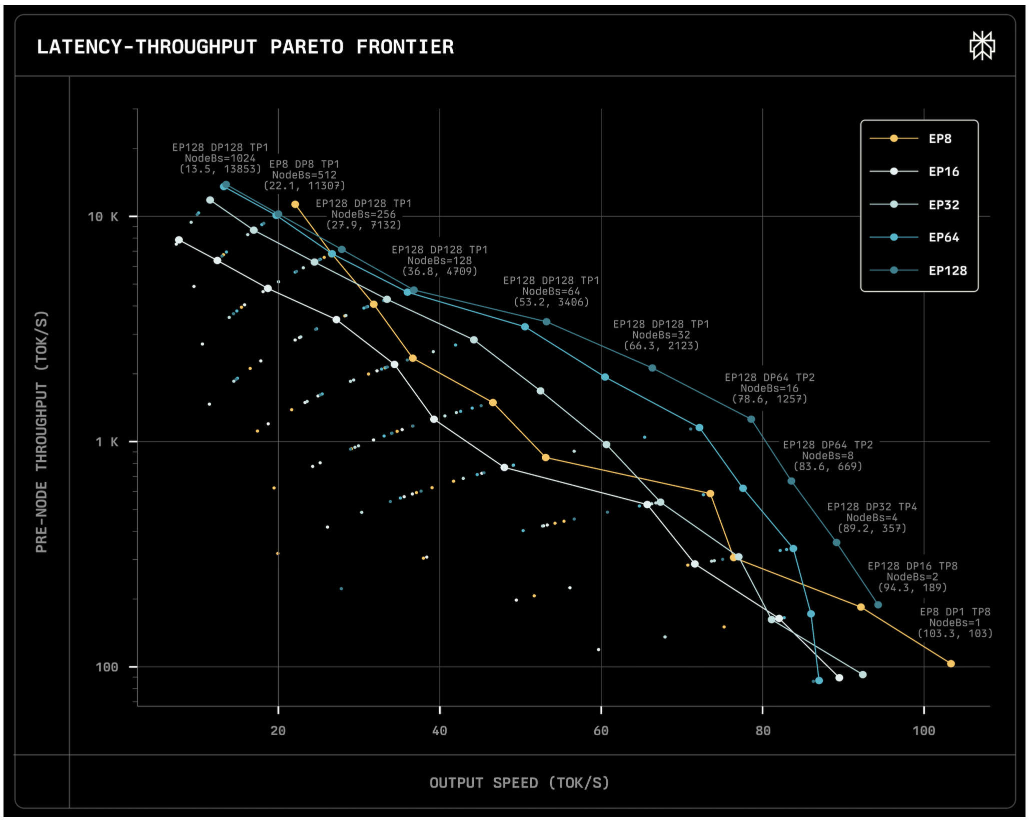
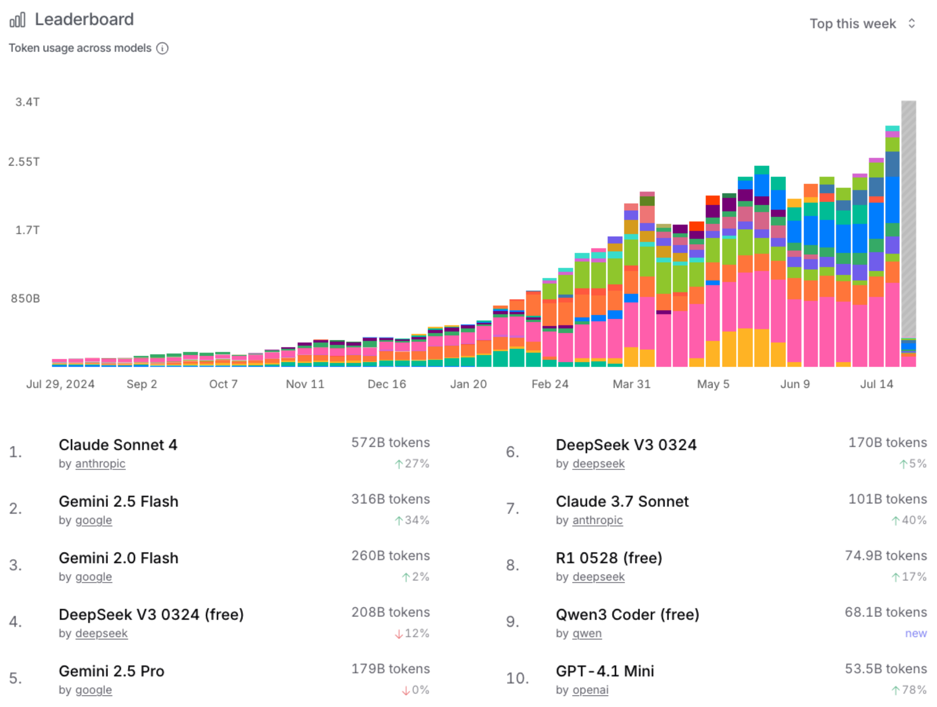


Figure 2: *Throughput per node by Perplexity*

Increasing the number of nodes operating within a single setup has a beneficial effect not only on the end-to-end performance but also on the per-node performance. In other words, we get an increased return on investment (ROI) on our fixed cost of hardware (GPUs). This benefit is described in a blog post by Perplexity (see also Fig. 2). Clearly, as we increase the number of nodes involved (the EP number), the per-node performance increases.

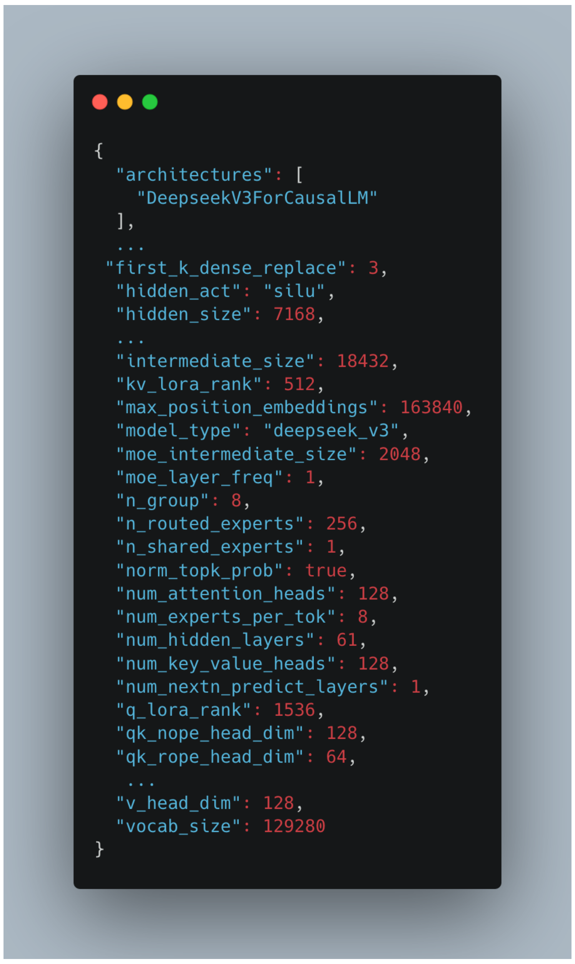


1.	Claude Sonnet 4 by anthropic	572B tokens ↑27%	6.	DeepSeek V3 0324 by deepseek	170B tokens ↑5%
2.	Gemini 2.5 Flash by google	316B tokens ↑34%	7.	Claude 3.7 Sonnet by anthropic	101B tokens ↑40%
3.	Gemini 2.0 Flash by google	260B tokens ↑2%	8.	R1 0528 (free) by deepseek	74.9B tokens ↑17%
4.	DeepSeek V3 0324 (free) by deepseek	208B tokens ↓12%	9.	Qwen3 Coder (free) by qwen	68.1B tokens new
5.	Gemini 2.5 Pro by google	179B tokens ↓0%	10.	GPT-4.1 Mini by openai	53.5B tokens ↑78%

Figure 3: *OpenRouter model popularity ranking, showing DeepSeekV3 as the most widely used open-source model (28.07.2025)*

The DeepSeek architecture is summarized by the model configuration available on Hugging Face (see Fig. 4). It consists of 61 layers (num-hidden-layers), three of which are dense (*first-k-dense-replace*), and the remaining 58 are MoE layers. Each MoE layer contains a modified self-attention mechanism (multi-head latent attention, or MLA), a gating mechanism, and 257 experts - 1 shared expert and 256 routed experts - as defined by the *n-shared-experts* and *n-routed-experts*. The MoE layers are followed by a traditional language modeling (LM) head. The DeepSeek team also proposed a *multi-token prediction* (MTP) head for speculative decoding. However, as modeling its real-world performance is complex and less relevant for larger batches, we exclude it from this analysis.

Each layer consists of a Multi-head Latent Attention (MLA) followed by a DeepSeekMoE, as illustrated in Figure 5. MLA is a variant of traditional attention that compresses the KV cache using a linear algebra optimization. Instead of storing full key-value pairs like other models (such as Llama, Qwen, etc.), MLA stores only a compressed latent representation of size *kv-lora-rank + qk-rope-head-dim*. This reduces memory bandwidth requirements during token-by-token decoding, since less KV cache memory needs to be loaded to produce each token.



```
{
  "architectures": [
    "DeepseekV3ForCausalLM"
  ],
  ...
 "first_k_dense_replace": 3,
  "hidden_act": "silu",
  "hidden_size": 7168,
  ...
  "intermediate_size": 18432,
  "kv_lora_rank": 512,
  "max_position_embeddings": 163840,
  "model_type": "deepseek_v3",
  "moe_intermediate_size": 2048,
  "moe_layer_freq": 1,
  "n_group": 8,
  "n_routed_experts": 256,
  "n_shared_experts": 1,
  "norm_topk_prob": true,
  "num_attention_heads": 128,
  "num_experts_per_tok": 8,
  "num_hidden_layers": 61,
  "num_key_value_heads": 128,
  "num_nextn_predict_layers": 1,
  "q_lora_rank": 1536,
  "qk_nope_head_dim": 128,
  "qk_rope_head_dim": 64,
   ...
  "v_head_dim": 128,
  "vocab_size": 129280
}
```

Figure 4: *DeepSeekV3 Hugging Face configuration*

These optimizations compress the KV cache to 70KB per token - a 2-7x reduction compared to other models (Qwen3 32B: 163KB, Llama 405B: 516KB per token [11]). This compression directly translates to reduced memory bandwidth requirements and lower inference costs. We detail the computational mechanics of MLA later; the key insight is that this architectural choice fundamentally alters the economics of serving large language models, especially with long context (such as agentic) use cases.

Following the MLA is the DeepSeekMoE component (see Fig. 5). The routing mechanism uses a linear layer mapping from *hidden-size* to *n-routed-experts* to classify which experts are most relevant for each token based on its semantic content. Each token is individually routed to a different set of experts. It is a common confusion that the experts are selected at the sequence (or query) level; we want to highlight that this is not the case. DeepSeek selects the top eight experts per token, with routing scores serving as weights when combining expert outputs.

Each expert contains a standard MLP structure with SwiGLU activation: three linear layers (W1 and W3: *hidden-size* $\rightarrow$ *moe-intermediate-size*, W2: reverse). Crucially, the *moe-intermediate-size* (2048) is smaller than *hidden-size* (7168) - the opposite of traditional dense models like Llama 3.3 70B, where intermediate dimensions are 3.5x larger (28672 vs 8192). This compression reduces per-expert computational costs while maintaining model capacity through expert diversity.

Beyond the eight routed experts, every token also passes through a shared expert that provides base knowledge common to all inputs. This hybrid approach balances specialization with computational efficiency.

This architecture is repeated across all 58 MoE layers, followed by the LM head for next-token prediction. The key architectural innovations - MLA for memory efficiency and sparse expert activation - represent a fundamental shift from traditional dense transformers toward economically optimized inference. In the following sections, we analyze the computational details of MLA and MoE components, identifying the primary bottlenecks that determine serving costs and scaling limits.

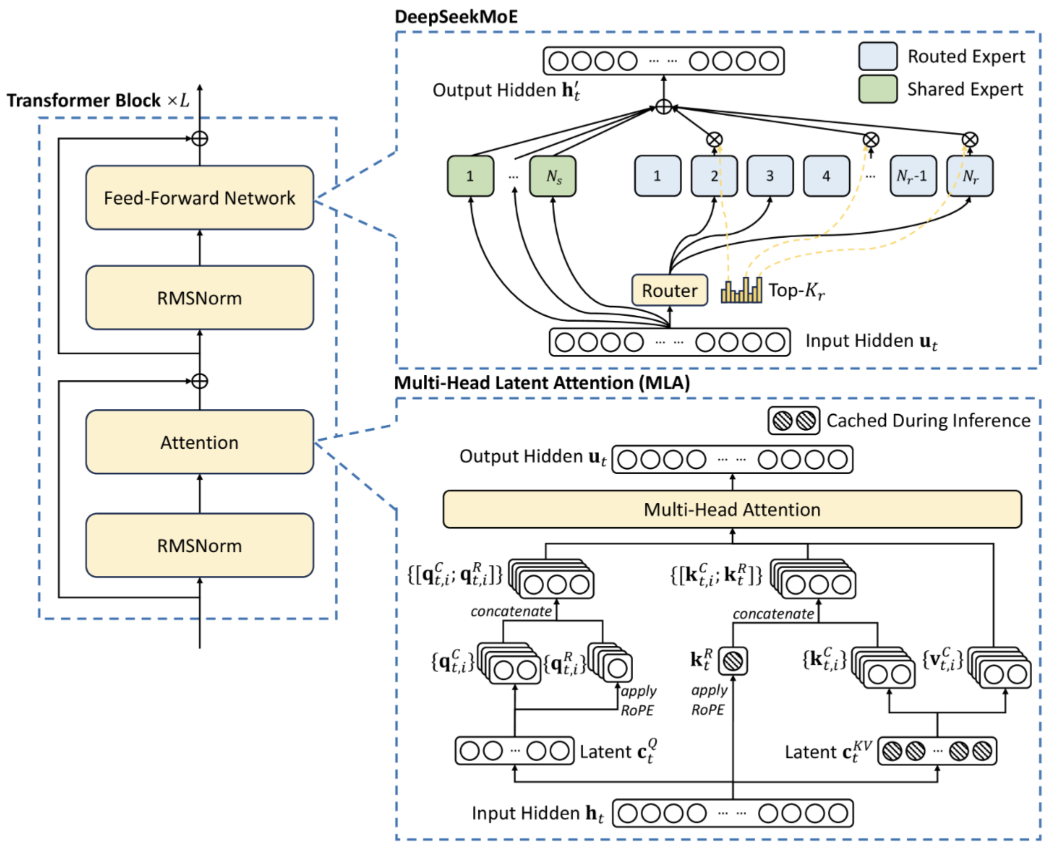


Figure 5: *DeepSeek v3 MoE transformer layers consist of 3 shared experts and 256 routed experts. The router, a single weight matrix, predicts scores where the top 8 scores are chosen and the token is routed to the corresponding expert. In total a token goes through 9 out of 257 (256 routed + 1 shared) experts in each forward pass. Figure taken from DeepSeek v3 [6]*

2 Inference Optimization Techniques

DeepSeek introduced a number of inference-time optimizations that substantially improve end-to-end performance. The optimizations demonstrate how the model was designed with hardware constraints in mind. We explore them one by one and later incorporate them into our theoretical model.

2.1 Expert Parallelism

As shown later, expert layers contain approximately 661B parameters, representing 98.5% of the total parameter count. This distribution necessitates careful consideration of parallelization strategies. To minimize weight-related overhead, parameter distribution rather than duplication provides the optimal approach.

In traditional tensor parallel configurations with dense FFN layers, communication involves dispatching and combining *hidden-size* values per token and layer. MoE models introduce complexity because batch tokens route to different model components (different experts). Given the compact dimensions of expert weight matrices ($d_{moe} = 2048$), tensor parallel sharding would fragment these matrices into excessively small components, resulting in suboptimal blocked matrix multiplication performance. Expert parallel sharding preserves matrix integrity, enabling more efficient memory access patterns during GEMM operations.

However, this approach increases total communication overhead to $d \times (n_r^E + n_s^E)$ per token and layer, where d is the model's hidden size. Because experts may reside on different devices, expert parallel distribution becomes more exposed to communication bottlenecks, fundamentally changing the performance characteristics compared to dense models. For deployments with small numbers of devices, particularly single or dual-node configurations, or those with very bad inter-node communication hardware, tensor parallel sharding can achieve superior performance due to the reduced communication overhead.

2.2 Expert Parallel Load Balancing

Expert routing probabilities exhibit non-uniform distributions (see Fig. 6), causing some experts to receive disproportionately higher request volumes. Naive uniform expert distribution across available devices creates two critical performance issues: (1) uneven communication patterns where bottlenecks stall entire forward passes, and (2) asymmetric computational loads across devices. Additionally, heavily utilized devices must handle increased activation loading and memory write-back operations, compounding performance degradation.

Load balancing strategies can mitigate these issues through intelligent expert distribution across devices to achieve more uniform computational loads. Furthermore, frequently accessed experts can be duplicated to reduce communication peaks, though this approach carries the trade-off of increased weight loading and reduced per-GPU KV-cache capacity. Since the expert layer computation is most often memory bound, having the same number of experts on each device is optimal. Therefore, the number of additional experts per layer must be constrained to multiples of the expert parallel size.

An interesting use case are uneven node configurations, where redundant experts can be used to fill up underutilized devices to achieve balanced expert distribution. For instance, the SGLang team reported using nine nodes for decode operations (72 expert parallel size) with 32 additional experts, achieving favorable trade-offs between additional memory overhead and reduced communication peaks [9].

Importantly, expert load balancing becomes increasingly challenging as node count increases. This degradation occurs because fewer nodes concentrate more experts per device, increasing the probability of achieving system-wide balance. So for deployments on very few nodes, the improvement in expert balancedness is not worth the corresponding redundant experts.

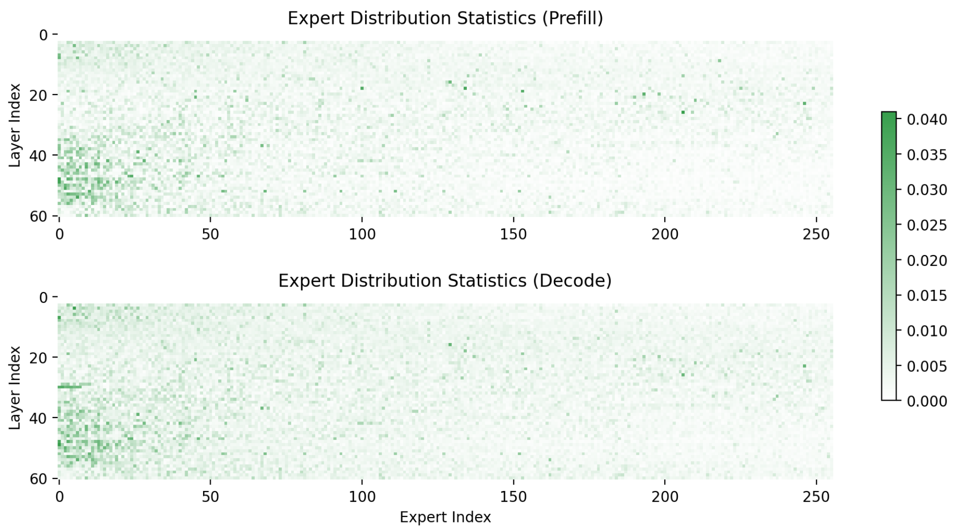


Figure 6: *Expert imbalance during prefill and decode phases as reported by SGLang. These are empirical observations from running over some specific datasets, so in your particular application the exact distribution will differ. The meta point, that expert usage is not uniformly distributed, stands, however.*

2.3 Location-Aware Expert Selection

To minimize the limiting inter-node communication, expert selection can incorporate locality penalties that preferentially route tokens to experts residing on the same node where their attention computations were performed. This approach reduces cross-node communication overhead, which often represents a primary bottleneck in distributed MoE inference.

During training, DeepSeek v3 [6] implemented expert routing constraints ensuring each token routes to at most M nodes. Node selection follows the sum of the highest $\frac{K_r}{M}$ affinity scores for experts distributed on each node, where K_r represents the number of routed experts and M the maximum number of nodes per token.

This methodology can be adapted for inference scenarios but requires careful tuning to balance locality benefits against response quality. A notable consequence of this approach is that token routing patterns become dependent on the position of the sequence within the batch, potentially creating position-dependent expert utilization patterns that may affect model responses.

2.4 Data-Parallel Attention

Attention computation employs a data-parallel approach, distributing requests across available devices (see Fig. 7). This strategy enables KV-cache sequences to remain on single devices, eliminating the need for duplication or inter-device communication of the latent KV-cache, which, on the other hand, would be required under tensor-parallel MLA computation due to projection matrices. However, this data-parallel approach requires duplicating all MLA weights across devices, approximately 10 GB for DeepSeek v3, and they must be loaded during each forward pass. This presents scalability tradeoffs for large-scale deployments. Specifically, in configurations exceeding 64 GPUs, the MLA weight parameters can consume greater memory resources than the expert layers themselves. This duplication reduces available KV-cache capacity and renders MLA computations memory-bound for most batch sizes.

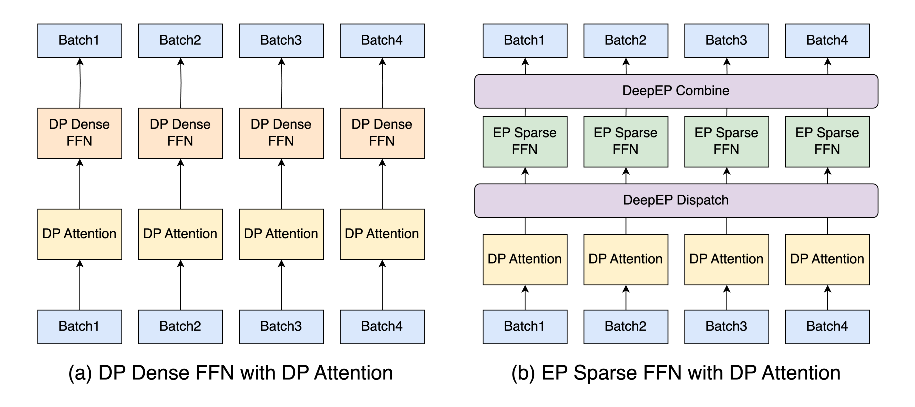


Figure 7: *Visualization of DP Attention from SGLang blog post. Different GPUs process different microbatches. It works both for dense layers (first three layers in DeepSeek) and mixture-of-experts layers*

2.5 Hiding Communication with Two-Batch Overlap

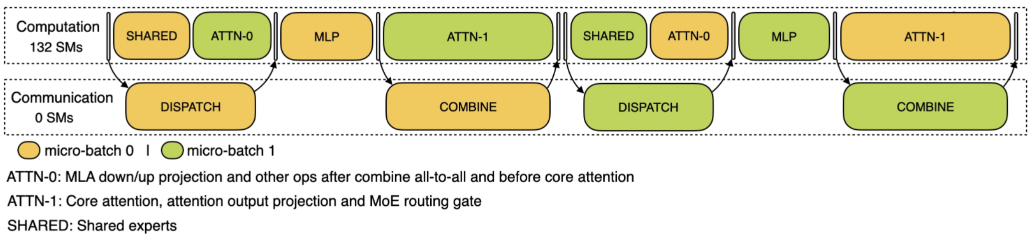

Computation 132 SMs
SHARED
ATTN-0
MLP
ATTN-1
Communication 0 SMs
DISPATCH
COMBINE
micro-batch 0 | micro-batch 1
ATTN-0: MLA down/up projection and other ops after combine all-to-all and before core attention
ATTN-1: Core attention, attention output projection and MoE routing gate
SHARED: Shared experts

Figure 8: *Creating two mini-batches allows the overlapping of the communication of one mini-batch with the computation of the other, thus in most cases hiding the communication from the overall runtime. Depending on the inference regime, different overlapping structures can be used. The figure depicts DeepSeek's setup for the decode phase during inference. Figure taken from DeepSeek's profiling of v3 [3]*

As shown previously, expert parallelism in MoE models generates approximately nine times the communication volume compared to traditional tensor parallelism. To mitigate this overhead, a two-batch overlap (TBO) strategy can be implemented to mask communication time behind computation. This approach partitions the global batch size into two micro-batches, enabling simultaneous execution where one micro-batch performs computation while the other handles communication operations. Effective overlap implementation requires careful orchestration of computational and communication phases. Figure 8 illustrates a basic TBO configuration for decode operations. Since communication operations consume minimal computational resources, TBO can achieve runtime improvements of close to a factor of two in certain scenarios.

2.6 Prefill-Decode Disaggregation

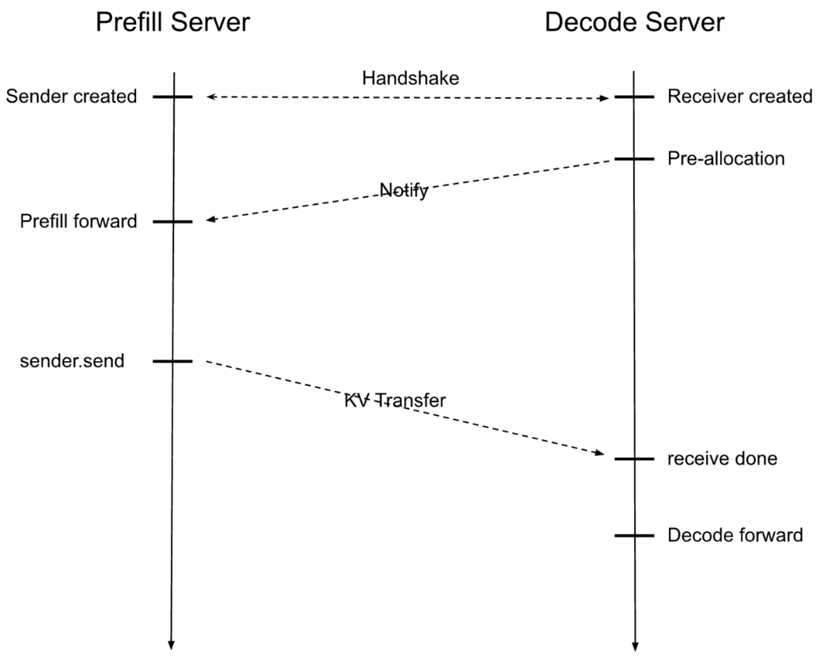


Figure 9: *Upon receiving an input request, the workflow proceeds as follows: 1) A Prefill Server and a Decode Server pair via a handshake, establishing a local sender and receiver, respectively. 2) The Decode Server pre-allocates the KV cache, signaling the Prefill Server to begin the model forward pass and compute the KV caches. 3) Once computed, the data transfers to the Decode Server, which handles iterative token generation. Cf. SGLang blog post*

LLM inference comprises two distinct phases with fundamentally different computational characteristics. Prefill operations process entire input sequences simultaneously, creating compute-intensive workloads with high FLOP utilization but minimal KV-cache requirements. Decode operations generate tokens iteratively, resulting in memory-bandwidth-bound computations. Also, decoding is much more latency sensitive due to its repetitive nature.

As explained in more detail on SGLang's blog post [9], traditional unified engines process prefill and decode batches together, introducing three critical inefficiencies: (1) incoming prefill batches interrupt ongoing decode operations, causing substantial token generation delays; (2) data-parallel attention imbalances occur when workers simultaneously handle different batch types, increasing decode latency; and (3) incompatibility arises with advanced expert placement strategies that require different dispatch modes for each phase.

Prefill-decode disaggregation resolves these issues by separating workloads into dedicated clusters optimized for each phase's requirements, with prefill usually needing fewer resources than decode due to better utilization.

3 Theoretical Performance Model

The theoretical performance model creates a virtual clone of our DeepSeek v3 model, enabling analysis of different hardware configurations to determine optimal MoE model serving strategies. Additionally, this framework allows identification of system bottlenecks across various deployment scenarios. While this model is specifically designed for DeepSeek v3 [6] and R1 [4], extensions to Kimi-K2 [7] and other MoE architectures are quite simple.

The theoretical performance model analyzes attention (MLA in the case of DeepSeek v3) and expert computations separately, as these components may be constrained by different resources at different times. Since two-batch overlap techniques may be employed to hide communication, these two parts of the model may operate without overlap and are thus not able to hide the memory loading over both computations combined. Furthermore, the model accounts for communication across potentially heterogeneous networks incorporating different intra- and inter-node communication hardware. Communication time can be optionally overlapped using TBO. Finally, we consider scenarios where expert distribution is nonhomogeneous, resulting in imbalanced communication, increased memory loading, and uneven computation across GPUs, which can bottleneck the entire system.

With communication computation overlap, we define the total system performance as:

$$t_{total} = \max(t_{\text{bottleneck\_MLA}} + t_{\text{bottleneck\_EP}}, t_{\text{communication}})$$

Without TBO, we can (1) look at the memory loading time and compute time for each block (EP and attention) and take the maximum; (2) add the communication time; and (3) consider imbalanced experts:

$$t_{total} = t_{\text{bottleneck\_MLA}} + t_{\text{bottleneck\_EP}} + t_{\text{communication}}$$

The model operates under the following assumptions:

- No computational and memory loading overhead from the DeepEP [10] TBO communication library. This assumption does not hold in practice, as the library launches a non-negligible number of CUDA kernels.
- All computations and weights are performed and stored in FP8, with the exception of communication operations, where dispatch occurs in BF16. This assumption is largely accurate since over 98% of parameters reside in expert weights, which utilize 8-bit quantization.
- The analysis focuses exclusively on decode performance without considering prefill operations. Theoretical derivations for prefill performance will be presented.

First lets look at the execution times of the MLA and expert MLP networks of a transformer block based on the operations performed and the memory loaded. Second we consider the communication. For reference all variable names are listed in Table 1.

Symbol	Size	Name	Name in Config
d	7168	Hidden Size	hidden-size
B		Global Batch Size	
S		Sequence Length	
β_{eb}		Expert Balancedness	
n_h	128	Number of Attention Heads	num-attention-heads
d_h	128	Per-Head Dimension	v-head-dim
d_c	512	KV Compression Dimension	kv-lora-rank
d'_c	1536	Query Compression Dimension	q-lora-rank
d_h^R	64	RoPE Per-Head Dimension	qk-rope-head-dim
n_r^E	256	Number of Routed Experts	n-routed-experts
$n_{r,u}^E$	8	Number of Routed Experts per Token	num-experts-per-tok
n_s^E	1	Number of Shared Experts	n-shared-experts
$n_{s,u}^E$	1	Number of Shared Experts per Token	
L_E	58	Number of Hidden Expert Layers Length	
d_{moe}	2048	Hidden Dimension of Each Expert	moe-intermediate-size
d_{dense}	18432	Hidden Dimension of Dense MLP	intermediate-size
L_D	3	Number of Hidden Dense Layers Length	first-k-dense-replace
L_{total}	61	Number of Hidden Layers Length	num-hidden-layers

Table 1: *Nomenclature of the symbols used in this description for DeepSeek v3, including the name in the publicly available config.json file.*

3.1 Memory Loading

To estimate the time spent loading from memory, we look at what gets loaded during each forward pass. First let's look at the MLA secondly at the expert networks.

3.1.1 MLA

The memory requirements during inference for the MLA can be categorized into three primary components: MLA weights (read), KV cache (read and write), and activations (write).

MLA weights are read once per forward pass. The MLA mechanism requires several weight matrices per layer, stored in FP8 format. As outlined for DeepSeek v2 [5], during inference, the up-projection matrices for the K- and V-tensors can be included into other matrices, thus reducing the overall number of matrices.

- $W^{DQ} \in \mathbb{R}^{d'_c \times d} = 1536 \times 7168 = 11.0\text{MB}$
- $W^{UQ} \in \mathbb{R}^{d_h \times n_h \times d'_c} = 128 \times 128 \times 1536 = 25.2\text{MB}$
- $W^{QR} \in \mathbb{R}^{d_h^R \times n_h \times d'_c} = 64 \times 128 \times 1536 = 12.6\text{MB}$
- $W^{DKV} \in \mathbb{R}^{d_c \times d} = 512 \times 7168 = 3.7\text{MB}$
- $W^{UK} \in \mathbb{R}^{d_h \times n_h \times d_c} = 128 \times 128 \times 512 = 8.4\text{MB}$
- $W^{UV} \in \mathbb{R}^{d_h \times n_h \times d_c} = 128 \times 128 \times 512 = 8.4\text{MB}$
- $W^{KR} \in \mathbb{R}^{d_h^R \times d} = 64 \times 7168 = 0.5\text{MB}$

- $W^O \in \mathbb{R}^{d \times d_h \times n_h} = 7168 \times 128 \times 128 = 117.4\text{MB}$

This yields a total of 187.2 MB per layer, resulting in 11.4 GB total attention weights that must be replicated across each data-parallel attention rank.

KV Cache size is significantly reduced in MLA architectures compared to traditional attention mechanisms. The cache size per token is determined by $(d_c + d_R^h) \times L$ elements, where $d_c = 512$ represents the KV compression dimension, $d_R^h = 64$ denotes the per-head dimension of decoupled queries and keys, and $L = 61$ is the number of layers.

The memory requirement becomes:

$$2 \times (d_c + d_R^h) \times L = 2 \times (512 + 64) \times 61 = 70\text{KB per token}$$

For a batch processing S input tokens and generating B output tokens, the system loads $S \times B$ tokens and saves B tokens to the cache.

Activation vectors require temporary storage for communication between expert computations. These activations are loaded once and written back once during the forward pass:

$$A = 2 \times B \times d \times L \approx B \times 0.9\text{MB}$$

While non-negligible for very large batch sizes, activation memory remains relatively small compared to weight and cache requirements.

3.1.2 Expert Networks

For the expert MLP networks, we have two sources of memory transfers: model weights, which get read once for each forward pass; and activations, which get loaded once in FP8 and written back once in BF16.

The latent vectors are loaded from and saved back to memory for communication. We load them once in FP8 and write them back once in BF16. This part is often negligible.

$$A = (1 + 2) \times B \times d \times L \approx B \times 1.4MB$$
$$A_{\text{total per GPU}} = (1 + 2) \times B \times d \times L \times \frac{1}{\beta_{eb}} \times \frac{n_{\text{shared\_experts\_used}} + n_{\text{routed\_experts\_used}}}{n\_GPUs}$$

The expert MLP is made up of two weight matrices W_1 and W_2, which perform a down-projection followed by an up-projection. Note that traditional transformer models have a larger intermediate space thus they first up-project, which makes the matrices much larger. Secondly, we have to account for the SwiGLU gate weight matrix as well. So the size of one expert is

$$W_1 = W_2 = d_{moe} \times d = 2048 \times 7168 \approx 14.7\text{MB}$$
$$W_{\text{Gate}} = d_{moe} \times d = 2048 \times 7168 \approx 14.7\text{MB}$$
$$W_E = W_1 + W_{\text{Gate}} + W_2 = 3 \times \text{moe\_intermediate\_size} \times d \approx 45\text{MB}.$$

Lastly, we have to account for the router in each expert which is of size $W_{\text{router}} = d \times \text{routed\_experts}$.

To get the expert weights per devices, the weights are distributed evenly over the devices. Additionally we assume that one device has to hold the full router (overestimation). So for each expert layer we get:

$$W_{\text{experts per GPU}} = n_{\text{layers}}^{\text{experts}} \times \left(W_E \times \lceil \frac{n_{\text{shared\_experts}} + n_{\text{routed\_experts}} + n_{\text{additional\_experts}}}{n\_GPUs} \rceil + W_{\text{router}} \right)$$

For DeepSeek v3 not all transformer layers are using the MoE architecture. The first three layers are made from traditional dense MLPs. Using d_{dense} as the intermediate size we get

$$\begin{aligned} W_1^d = W_2^d &= d_{dense} \times d = 18432 \times 7168 \approx 132\text{MB} \\ W_{\text{Gate}}^d &= d_{dense} \times d = 18432 \times 7168 \approx 132\text{MB} \\ W_{MLP}^d &= W_1^d + W_{\text{Gate}}^d + W_2^d = 3 \times d_{dense} \times d \approx 396\text{MB}. \end{aligned}$$

When serving in expert parallel with SGLang these weights are sharded data parallel. Thus

$$W_{\text{MLP per GPU}}^{\text{dense}} = n_{\text{layers}}^{\text{dense}} \times W_{MLP}^d \approx 3 \times 396\text{MB} \approx 1.2\text{GB}$$

Extension to Small Batch Sizes Since MoE models for small batch sizes do not need to load the full model, the question becomes at what point the full model is likely to be loaded in full once.

For small batch sizes, we need to consider only the activated weights. To find the batch size at which we can consider the full model to be loaded every time, we look at the distribution of sampling the experts. We can think of this as repeated urn experiments. In each experiment, we select $n_{\text{routed experts used}}$ balls from $n_{\text{routed experts}}$ without replacement, and repeat this experiment N times, where N is our batch size. If we consider each ball as a separate "type", then:

- Population size: $M = n_{\text{routed experts}}$, each ball type has exactly one item in the population
- Sample size: $k = n_{\text{routed experts used}}$ (drawn without replacement)

For each experiment, let X_i be the number of times ball i is selected (which is either 0 or 1 since each ball appears only once). Then:

$$(X_1, X_2, ..., X_M) \sim \text{Multivariate Hypergeometric}(M; 1, 1, 1, ..., 1; k)$$

For the full problem across N experiments (where N is our batch size), let Y_i be the total number of times ball i is selected across all N experiments, then $(Y_1, Y_2, ..., Y_M)$ follows a sum of N independent multivariate hypergeometric distributions. So the number of unique balls is:

$$U = |i : Y_i > 0| = \sum_i I(Y_i > 0),$$

where $I(\cdot)$ is the indicator function.

The expected value $\mathbb{E}[U]$ can be simplified to

$$\begin{aligned} \mathbb{E}[U] = \mathbb{E}[\sum_i I(Y_i > 0)] &= \sum_i 1 \times (P(Y_i > 0)) \\ &= \sum_i (1 - P(Y_i = 0)), \end{aligned}$$

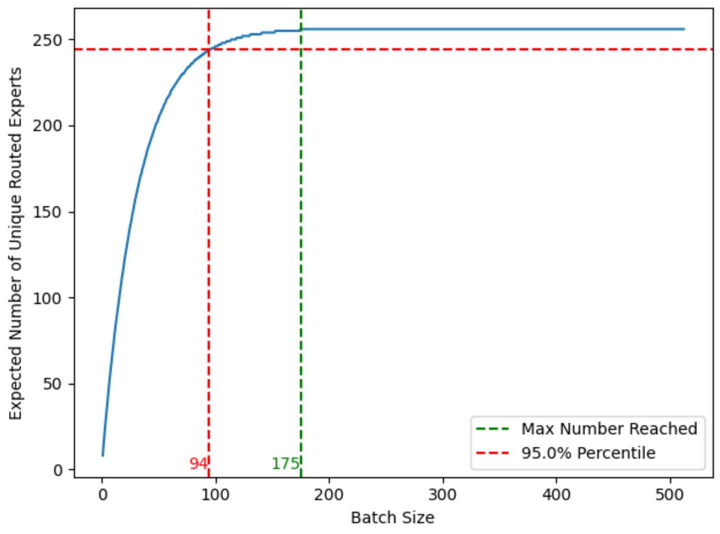


Figure 10: *Statistical model of activated experts at given batch sizes, assuming uniform expert activation. While this assumption does not strictly hold in practice, as some experts may remain inactive even across multiple forward passes at larger batch sizes, the approximation provides reasonable accuracy.*

where $P(Y_i = 0)$ stands for ball i (or in our case expert i) which was never selected. Calculating this:

$$P(\text{ball } i \text{ selected in experiment } j) = P(X_{i,j}) = \frac{k}{M}$$
$$P(Y_i = 0) = P(\text{ball } i \text{ never selected in } N \text{ experiments}) = (1 - 1/32)^N = (31/32)^N$$
$$\mathbb{E}[U] = 256(1 - (31/32)^N),$$

where the last equality on the first line stems from the fact:

$$P(X_{i,j}) = \frac{1}{M} + \frac{1}{M-1} * \frac{M-1}{M} + \ldots = \sum_{l=M-k}^{M} \frac{1}{M} = \frac{k}{M}.$$

Plotting the expected value for DeepSeek v3, as shown in Figure 10, shows that for a batch size greater than 128, we can assume the model is loaded in full. From this we can estimate the expected number of activated routed experts given a batch size.

Since we are calculating the bottleneck time for each device, we have to take into account not the average number of routed experts on each device but the maximum number. For this we use the asymptotic approximation of balls into bins problem. This should work fairly well given larger numbers of activated experts. Therefore we get:

$$\mathbb{E}\left[n^E_{max\ total}\right] = \frac{n^E_{active}}{n_{GPU}} + \sqrt{2 \times n^E_{active} \times \frac{\log(n_{GPU})}{n_{GPU}}},$$

where the number of active experts n^E_{active} is estimated as outlined above.

3.1.3 Embedding Layers

The embedding matrix size is $V \times d$, where V is the vocabulary size. As we need to embed and un-embed, and each embedding matrix is sharded across GPUs in a data-parallel fashion, the per-GPU size is:

$$W_{\text{E per GPU}} = 2 \times V \times d = 2 \times 129280 \times 7168 \approx 1.9\text{GB}$$

As this is a large matrix with $V \gg 100k$, we assume this calculation is always memory-bound, and thus we only take into account the memory loading time.

3.2 Computation

To quantify computational latency, we examine the multi-head latent attention (MLA) mechanism following the architectural specification detailed in DeepSeek v2 Appendix C [5]. Our analysis incorporates matrix absorption optimizations that enable certain linear transformations to be merged during inference. We validate our computational framework against the DeepSeek v3 training time calculator [1] to ensure consistency, though we need to employ some changes due to optimizations only possible during inference. Furthermore, the single-token generation characteristic of decode operations substantially simplifies several equations relative to training contexts where full sequence processing is required. We denote computations specific to prefill scenarios with *Prefill* annotations to distinguish where prefill diverges from decode execution paths.

Our computational analysis has three main parts: a vanilla MLA implementation baseline, optimized MLA with matrix absorption techniques, and expert network computational latency.

3.2.1 MLA

MLA computational demands differ substantially between prefill and decode phases due to sequence length scaling characteristics, necessitating phase-specific analysis of each MLA component. We begin by reviewing the MLA computation procedure as specified in DeepSeek v2 Appendix C [5]:

$$\begin{aligned}
\mathbf{c}_t^Q &= W^{DQ}\mathbf{h}_t, \\
\left[\mathbf{q}_{t,1}^C; \mathbf{q}_{t,2}^C; \ldots; \mathbf{q}_{t,n_h}^C\right] = \mathbf{q}_t^C &= W^{UQ}\mathbf{c}_t^Q, \\
\left[\mathbf{q}_{t,1}^R; \mathbf{q}_{t,2}^R; \ldots; \mathbf{q}_{t,n_h}^R\right] = \mathbf{q}_t^R &= \text{RoPE}(W^{QR}\mathbf{c}_t^Q), \\
\mathbf{q}_{t,i} &= \left[\mathbf{q}_{t,i}^C; \mathbf{q}_{t,i}^R\right], \\
\mathbf{c}_t^{KV} &= W^{DKV}\mathbf{h}_t, \\
\left[\mathbf{k}_{t,1}^C; \mathbf{k}_{t,2}^C; \ldots; \mathbf{k}_{t,n_h}^C\right] = \mathbf{k}_t^C &= W^{UK}\mathbf{c}_t^{KV}, \\
\mathbf{k}_t^R &= \text{RoPE}(W^{KR}\mathbf{h}_t), \\
\mathbf{k}_{t,i} &= \left[\mathbf{k}_{t,i}^C; \mathbf{k}_{t,i}^R\right], \\
\left[\mathbf{v}_{t,1}^C; \mathbf{v}_{t,2}^C; \ldots; \mathbf{v}_{t,n_h}^C\right] = \mathbf{v}_t^C &= W^{UV}\mathbf{c}_t^{KV}, \\
\mathbf{o}_{t,i} &= \sum_{j=1}^{t} \text{Softmax}_j\left(\frac{\mathbf{q}_{t,i}^T\mathbf{k}_{j,i}}{\sqrt{d_h + d_h^R}}\right)\mathbf{v}_{j,i}^C, \\
\mathbf{u}_t &= W^O\left[\mathbf{o}_{t,1}; \mathbf{o}_{t,2}; \ldots; \mathbf{o}_{t,n_h}\right],
\end{aligned}$$

where $\mathbf{c}_t^Q$, $\mathbf{c}_t^{KV}$ are the compressed Q and KV tensors respectively. During decode operations, MLA requires up-projecting the K- and V-tensors for every token within the cache, leading to significant computational overhead. To mitigate this burden, the up-projection matrices for K- and V-tensors can be absorbed into existing matrix operations, thereby reducing the total number of matrix-vector multiplications, as briefly touched upon in the DeepSeek v2 paper [5]. Through computational reordering, the self-attention mechanism, for instance, transforms to:

$$\begin{aligned} q_{t,i}^T k_{j,i} &= (W^{UQ}_{[i\times n_h:(i+1),:]} c_t^Q)^T W^{UK}_{[i\times n_h:(i+1),:]} c_j^{KV} + q_{t,i}^{R^T} k_{j,i}^R \\ &= c_t^{QT} (W^{UQ^T} W^{UK})_{[i\times n_h:(i+1),:]} c_{j,i}^{KV} + q_t^{R^T} k_{j,i}^R, \end{aligned}$$

The most important advantage, as demonstrated in the following sections, is that the KV-cache no longer requires up-projection operations. We apply analogous reordering to the V-tensor up-projection weight matrix W^{UV}, absorbing it into the output projection matrix W^O.

However, materializing $W^{UQ^T} W^{UK}$ increases the amount of memory transfer and reduces available KV-cache capacity. But there is a way to have our cake and eat it. Our approach diverges from DeepSeek's hint by avoiding materialization of the resulting composite matrix. Instead, we achieve efficiency through dynamic computation: rather than storing the composite matrix $W^{UQ^T} W^{UK}$, we compute it on-demand during each forward pass while calculating q_t^C. This strategy maintains an identical memory footprint and loading patterns while eliminating the computationally expensive sequence-length dependency imposed upon us due to the up-projections of K- and V-tensors during the decode phase.

To understand the computational requirements of MLA, we begin with analyzing the straightforward vanilla implementation to establish baseline FLOP counts, then progress to the optimized variant and show its performance improvements.

3.2.2 Vanilla MLA Implementation

The vanilla implementation follows the MLA specification from DeepSeek v2 Appendix C [5], comprising three phases: latent projection, self-attention, and output projection.

Latent Up-/Down-Projection The latent projection involves two sequential operations: down-projection to the latent space, followed by up-projection to the attention dimension. For the sake of simplicity, we ignore the RoPE and Softmax calculations.

Prefilling:

$$FLOPs_{\text{q\_down\_proj}} = 2 \times B \times S \times d \times d'_c \tag{1}$$

$$FLOPs_{\text{q\_up\_proj}} = 2 \times B \times S \times d'_c \times n_h \times d_h \tag{2}$$

$$FLOPs_{\text{q\_RoPE\_proj}} = 2 \times B \times S \times d'_c \times n_h \times d_h^R \tag{3}$$

$$FLOPs_{\text{kv\_down\_proj}} = 2 \times B \times S \times d \times d_c \tag{4}$$

$$FLOPs_{\text{k\_up\_proj}} = 2 \times B \times S \times n_h \times d_c \times d_h^{qk} \tag{5}$$

$$FLOPs_{\text{k\_RoPE\_proj}} = 2 \times B \times S \times d \times d_h^R \tag{6}$$

$$FLOPs_{\text{v\_up\_proj}} = 2 \times B \times S \times n_h \times d_c \times d_h^v \tag{7}$$

During decode operations, most sequence-length dependencies can be eliminated since intermediate values are either used once or cached. However, performing K- and V-tensor projections for each token in the context remains necessary due to caching in latent format. This creates a problematic sequence-length dependency that significantly degrades performance. So vanilla decode projection becomes **Decode (single token)**:

$$
\begin{aligned}
FLOPs_{\text{q\_down\_proj}} &= 2 \times B \times d \times d'_c && (8)\\
FLOPs_{\text{q\_up\_proj}} &= 2 \times B \times d'_c \times n_h \times d_h && (9)\\
FLOPs_{\text{q\_RoPE\_proj}} &= 2 \times B \times d'_c \times n_h \times d_h^R && (10)\\
FLOPs_{\text{kv\_down\_proj}} &= 2 \times B \times d \times d_c && (11)\\
FLOPs_{\text{k\_up\_proj}} &= 2 \times B \times S \times n_h \times d_c \times d_h^{qk} && (12)\\
FLOPs_{\text{k\_RoPE\_proj}} &= 2 \times B \times d \times d_h^R && (13)\\
FLOPs_{\text{v\_up\_proj}} &= 2 \times B \times S \times n_h \times d_c \times d_h^v, && (14)
\end{aligned}
$$

where the $FLOPs_{\text{k\_RoPE\_proj}}$ only need to be calculated once for each k, as they are cached.

Attention Computation The attention mechanism computes query-key interactions against cached key-value pairs, exhibiting computational complexity that scales with sequence length.

Prefilling:

$$
\begin{aligned}
FLOPs_{\text{qk\_scores}} &= 2 \times B \times S^2 \times n_h \times (d_h + d_h^R) && (15)\\
FLOPs_{\times\text{v}} &= B \times S^2 \times n_h \times d_h && (16)
\end{aligned}
$$

Decode:

$$FLOPs_{\text{qkv}} = B \times n_h \times S \times (2 \times (d_h + d_h^R) + d_h) \tag{17}$$

Output Linear Transformation The final linear transformation projects attention outputs back to the model's hidden dimension:

Prefilling:

$$FLOPs_{\text{out\_lin}} = 2 \times B \times S \times n_h \times d_h \times d \tag{18}$$

Decode:

$$FLOPs_{\text{out\_lin}} = 2 \times B \times n_h \times d_h \times d \tag{19}$$

We now examine the computational modifications when implementing the non-materialized matrix-absorption approach.

3.2.3 Matrix-Absorbed MLA Implementation

The primary goal of the two matrix absorptions (W^{UK} into W^{UQ} and W^{UV} into W^O) is to eliminate per-token KV-cache up-projections by enabling direct computation on compressed KV-tensors. This fundamentally alters the memory access pattern.

Latent Up-/Down-Projection Since we absorb the up-projection matrices for K- and V-tensors, we no longer need to perform these projections.

Prefilling:

$$FLOPs_{\text{q\_down\_proj}} = 2 \times B \times S \times d \times d'_c \tag{20}$$
$$FLOPs_{\text{q\_RoPE\_proj}} = 2 \times B \times S \times d'_c \times n_h \times d_h^R \tag{21}$$
$$FLOPs_{\text{kv\_down\_proj}} = 2 \times B \times S \times d \times d_c \tag{22}$$
$$FLOPs_{\text{k\_RoPE\_proj}} = 2 \times B \times S \times d \times d_h^R \tag{23}$$
$$\tag{24}$$

Decode (single token):

$$FLOPs_{\text{q\_down\_proj}} = 2 \times B \times d \times d'_c \tag{25}$$
$$FLOPs_{\text{q\_RoPE\_proj}} = 2 \times B \times d'_c \times n_h \times d_h^R \tag{26}$$
$$FLOPs_{\text{kv\_down\_proj}} = 2 \times B \times d \times d_c \tag{27}$$
$$FLOPs_{\text{k\_RoPE\_proj}} = 2 \times B \times d \times d_h^R, \tag{28}$$

This eliminates the sequence-length dependency during the projection stage, which constitutes a significant computational bottleneck.

Attention Computation In this self-attention computation, we need to take the absorbed $W^{UQ^T} W^{UK}$ into account.

Prefilling:

$$FLOPs_{\text{q\_}W^{UQ^T}} = 2 \times B \times S \times n_h \times d_h \times d'_c \tag{29}$$
$$FLOPs_{\times W^{UK}} = 2 \times B \times S \times n_h \times d_h \times d_c \tag{30}$$
$$FLOPs_{\text{qk\_scores}} = 2 \times B \times S^2 \times n_h \times (d_c + d_h^R) \tag{31}$$
$$FLOPs_{\times \text{v}} = B \times S^2 \times n_h \times d_c \tag{32}$$

Decode:

$$FLOPs_{\text{q\_}W^{UQ^T}} = 2 \times B \times n_h \times d_h \times d'_c \tag{33}$$
$$FLOPs_{\times W^{UK}} = 2 \times B \times n_h \times d_h \times d_c \tag{34}$$
$$FLOPs_{\text{qk\_scores}} = 2 \times B \times S \times n_h \times (d_c + d_h^R) \tag{35}$$
$$FLOPs_{\times \text{v}} = B \times S \times n_h \times d_c, \tag{36}$$

As shown, the sequence-length dependency is eliminated for $FLOPs_{\times W^{UK}}$ and $FLOPs_{\times W^{UV}}$ without introducing additional matrix materialization.

Output Linear Transformation The final linear transformation again projects attention outputs to the model dimension, incorporating the absorption of W^{UV} into W^O. We again avoid materializing the absorbed matrix to minimize memory overhead.

Prefilling:

$$FLOPs_{\text{out\_lin}} = 2 \times B \times S \times (n_h \times d_h \times d + n_h \times d_h \times d_c) \tag{37}$$

Decode:

$$FLOPs_{\text{out\_lin}} = 2 \times B \times (n_h \times d_h \times d + n_h \times d_h \times d_c) \tag{38}$$

3.2.4 Expert Networks

Following the MLA computational analysis, expert-network computation is relatively simple. Each expert module comprises two components: (1) a router and (2) the experts themselves. The experts consist of two layers with inverse dimensions and a SwiGLU activation function, incorporating gate projection weights of matching dimensions.

$$\text{Down-projection: } B \times 2 \times d \times d_{\text{moe}} \tag{39}$$
$$\text{Up-projection: } B \times 2 \times d \times d_{\text{moe}} \tag{40}$$
$$\text{Gate projection: } B \times 2 \times d \times d_{\text{moe}} \tag{41}$$
$$\tag{42}$$

In total, for all experts, we get:

$$FLOPs_{\text{linear\_layers}} = 3 \times 2 \times B \times d \times d_{\text{moe}} \tag{43}$$
$$FLOPs_{\text{router}} = 2 \times B \times d \times n_r^E \tag{44}$$
$$FLOPs_{\text{moe\_total}} = FLOPs_{\text{linear\_layers}} \times (n_s^E + n_{r,u}^E) + FLOPs_{\text{router}}. \tag{45}$$

With expert imbalance (characterized by the expert imbalance factor β_{eb} that we introduce in later sections), $FLOPs_{\text{moe\_total}} = FLOPs_{\text{linear\_layers}} \times \frac{1}{\beta_{eb}} \times \frac{n_{s,u}^E + n_{r,u}^E}{n\_GPUs} + \frac{FLOPs_{\text{router}}}{n\_GPUs}$

3.3 Communication

3.3.1 Communication Base Model

Following the analysis presented in the SGLang blog post [9], the only inter-GPU communications stem from expert-parallelism sharding. Figure 11 illustrates this communication pattern during forward passes within individual layers. Each layer has two distinct communication phases: a dispatch phase routing tokens from data-parallel MLA blocks to selected experts, followed by a combine phase aggregating expert computation results for propagation to the next data-parallel MLA block in the subsequent layer.

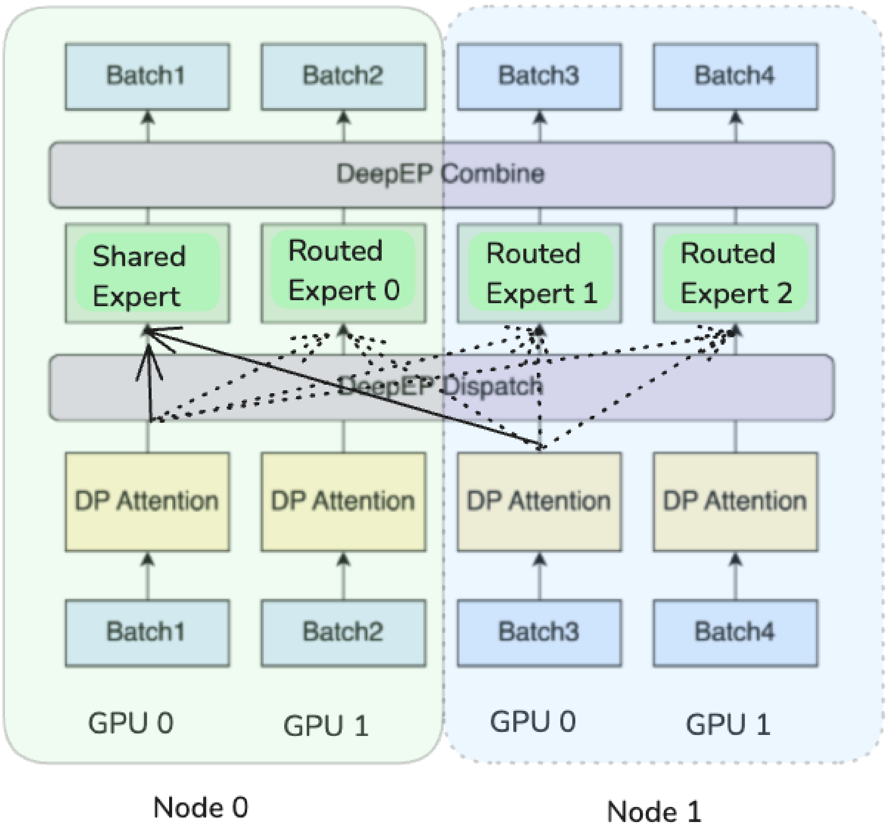


Figure 11: *Communication pattern during expert-parallel sharding. Each GPU sends out* $n^E_{r,u} + n^E_{s,u} = 9$ *messages going from the MLA to the expert computation and receives* $n^E_{r,u} + n^E_{s,u} = 9$ *many messages going from the experts computation to the next MLA step. This communication can happen intra- or inter-node, where systems with larger EP size increase the percentage of inter-node communication. The base image was taken from the SLGang blog post [9].*

The DeepSeek scaling v3 analysis [11] establishes a baseline communication model assuming uniform expert distribution across devices, where "each device holds one expert's parameters and processes approximately 32 tokens at a time". This configuration corresponds to a 257-GPU deployment with homogeneous network connections (each GPU-to-GPU connection has the same bandwidth and latency). We extend their formulation to accommodate arbitrary system configurations under assumptions of uniform network topology and perfectly balanced expert allocation. For each GPU communication link, each dispatch and combine operation processes $\frac{B}{2\times n_{\text{GPUs}}}$ tokens (accounting for dual-batch overlap), with replication across $n^E_{total,u} = n^E_{r,u} + n^E_{s,u}$. The total communication volume per GPU link per forward pass for DeepSeek v3 architectures becomes:

$$\begin{aligned}\text{comm\_amount}_{\text{v3 / r1}} &= \underbrace{2}_{\substack{\text{2b-overlap}\\ \rightarrow 2\ \times\text{total time}}} \times \underbrace{(1\text{Byte} + 2\text{Bytes})}_{\substack{\text{dispatch in FP8 (1 byte),}\\ \text{combine in BF16 (2 bytes)}}} \times \frac{B}{2 \times n_{\text{GPUs}}} \times n^{E}_{total,u} \times d \times L \\ &= 2 \times \frac{B}{2 \times n_{\text{GPUs}}} \times 3 \times (8+1) \times 7168 \times 61 \\ &= \frac{B}{n_{\text{GPUs}}} \times 0.0118\ \text{GB}.\end{aligned}$$

The leading factor of 2 reflects computation-communication micro-batch overlap, where consecutive batch processing introduces sequential dependencies. Dispatch operations utilize FP8 precision while combine phases use BF16 precision.

The effective communication bandwidth corresponds to the minimum link bandwidth within the system topology. In well-balanced configurations without bottlenecks this is the link to each GPU. In general cases the constraint becomes:

$$\text{comm\_bandwidth} = \min(\text{comm\_bandwidth}_{\text{GPU}}, \frac{\text{comm\_bandwidth}_{\text{system bottleneck}}}{n_{\text{GPUs using bottleneck}}}).$$

Within homogeneous network environments, the communication execution time follows:

$$t_{\text{comm}} = \text{comm\_amount}_{\text{v3 / r1}} \times \frac{1}{\text{comm\_bandwidth}}.$$

3.3.2 Improvement 1: Non-Heterogeneous Communication Links

Standard systems often have different interconnect speeds for intra- and inter-node communication. Given that $\frac{1}{n_{\text{nodes}}}$ of total communication volume remains within individual nodes, this fraction of communication becomes negligible in heterogeneous network configurations where interconnect speeds differ substantially (for example, NVLink at 450 GB/s versus InfiniBand at 50 GB/s). The NVL72 rack configuration represents a notable exception, providing uniform NVLink connectivity across all nodes within the rack.

For systems with heterogeneous interconnections, the total communication time becomes:

$$\begin{aligned}t_{\text{comm}} = \text{comm\_amount}_{\text{v3 / r1}} \times max(&\frac{n_{\text{nodes}} - 1}{n_{\text{nodes}}} \times \frac{1}{\text{comm\_bandwidth}_{\text{inter-node}}}, \\ &\frac{1}{n_{\text{nodes}}} \times \frac{1}{\text{comm\_bandwidth}_{\text{intra-node}}})\end{aligned}$$

3.3.3 Improvement 2: Expert Imbalance

Our initial analysis assumed uniform expert distribution across GPUs, enabling straightforward communication volume calculations from the data-parallel MLA perspective. However, this assumption breaks down in practical deployments. As an easy counter example, consider a system deploying 9 experts across 8 GPUs: one GPU must accommodate 2 experts due to the discreteness of the experts, resulting in nearly double the communication overhead compared to balanced configurations:

$$2 \times \underbrace{2}_{\substack{\text{2b-overlap} \\ \rightarrow 2\ \times\text{total time}}} \times \underbrace{(1\text{Byte} + 2\text{Bytes})}_{\substack{\text{dispatch in FP8 (1 byte),} \\ \text{combine in BF16 (2 bytes)}}} \times \frac{B}{2} \times d \times L = 2 \times B \times \frac{1}{8+1} \times 0.0118 \text{ GB}$$

$$> \frac{B}{8} \times 0.0118 \text{ GB}$$

While sufficiently large batch sizes with random expert routing (and appropriate shared expert replication) could theoretically rebalance this load, empirical measurements in production systems show inherent differences in expert utilization that contradicts this assumption.

From the perspective of individual GPUs hosting experts, the communication volume becomes:

$$\underbrace{2}_{\substack{\text{2b-overlap} \\ \rightarrow 2\ \times\text{total time}}} \times \underbrace{(1\text{Byte} + 2\text{Bytes})}_{\substack{\text{dispatch in FP8 (1 byte),} \\ \text{combine in BF16 (2 bytes)}}} \times \frac{B}{2} \times d \times L \times \frac{n^E_{total,u}}{n_{\text{GPUs}}}$$

Incorporating the expert load imbalance factor (introduced in the next section) into the formulation yields:

$$\text{comm\_amount}_{\text{v3 / r1}} = B \times (1+2) \times d \times L \times \frac{1}{\beta_{eb}} \times \frac{n^E_{total,u}}{n_{\text{GPUs}}}$$

For heterogeneous network configurations, the resulting communication time then becomes:

$$t_{\text{comm}} = \text{comm\_amount}_{\text{v3 / r1}} \times \max\left(\frac{n_{\text{nodes}} - 1}{n_{\text{nodes}}} \times \frac{1}{\text{comm\_bandwidth}_{\text{inter-node}}}, \frac{1}{n_{\text{nodes}}} \times \frac{1}{\text{comm\_bandwidth}_{\text{intra-node}}}\right)$$

3.3.4 List of Common Interconnects

Table 2 presents unidirectional bandwidth specifications for all-to-all communication patterns for commonly used interconnect technologies, where communication throughput is constrained by the bandwidth available to individual GPUs (unidirectional bandwidth):

System	Per Link Bandwidth
InfiniBand (IB) Fully connected CX7 400Gbps	50 GB/s
InfiniBand (IB) Single port connected CX7 400Gbps	25 GB/s
NVLink H100/H200	450 GB/s
NVLink B200	900 GB/s

Table 2: *Unidirectional bandwidth specifications across different interconnect technologies*

3.4 Expert Balancedness

The distribution of the experts over the GPUs can have a big impact on the communication and execution times of the expert layers. As an illustrative example, we can take a system with 2x8 H100 GPUs and distribute all the experts uniformly. In this case, there is a GPU which has the shared expert plus roughly 16 routed ones. Since all items in a batch will go to the shared expert, this GPU has to load roughly $\frac{1+\frac{n_{\text{shared\_experts\_used}}+n_{\text{routed\_experts\_used}}}{n_{\text{GPUs}}}}{\frac{n_{\text{shared\_experts\_used}}+n_{\text{routed\_experts\_used}}}{n_{\text{GPUs}}}} \approx 2,7\times$ more activations than the other GPUs. Furthermore, 2.7 times more communication volume will go through the link connecting to the GPU.

To model this imbalance, we define and expose the variable β_{eb} to the user. Similar to the definition of SGLang, β_{eb} is defined as the ratio between mean expert load and maximum expert load among GPUs, so:

$$\beta_{eb} = mean_{layers}\left(\frac{\text{avg routed request per layer}}{\text{max routed request (of any GPU) per layer}}\right)$$

Therefore, $\beta_{eb\_gpu} = 1$ is a balanced case and $\beta_{eb} = \frac{1}{n_{\text{GPUs}}}$ would be completely imbalanced. Thus the average load increases by $L_{\text{imbalanced}} = \frac{1}{\beta_{ep}} L_{\text{balanced}}$. For balancing the experts, some $n_{\text{additional\_experts}}$ get duplicated onto multiple GPUs. This can lead to an increase in EP memory loading time. As EP is often memory-bound, this can lead to an increased EP execution time. Thus balancing the experts is a tradeoff between loading more weights and more homogeneous communication and computation. Finally, one has to ensure that $(n_{\text{routed\_experts}} + n_{\text{additional\_experts}}) \bmod ep_{\text{size}} = 0$, as otherwise imbalance in the memory loading and computations would be introduced by design.

Figure 12 from the SGLang blog post [9] shows examples of expert balancedness given a number of GPUs and potentially active load balancing.

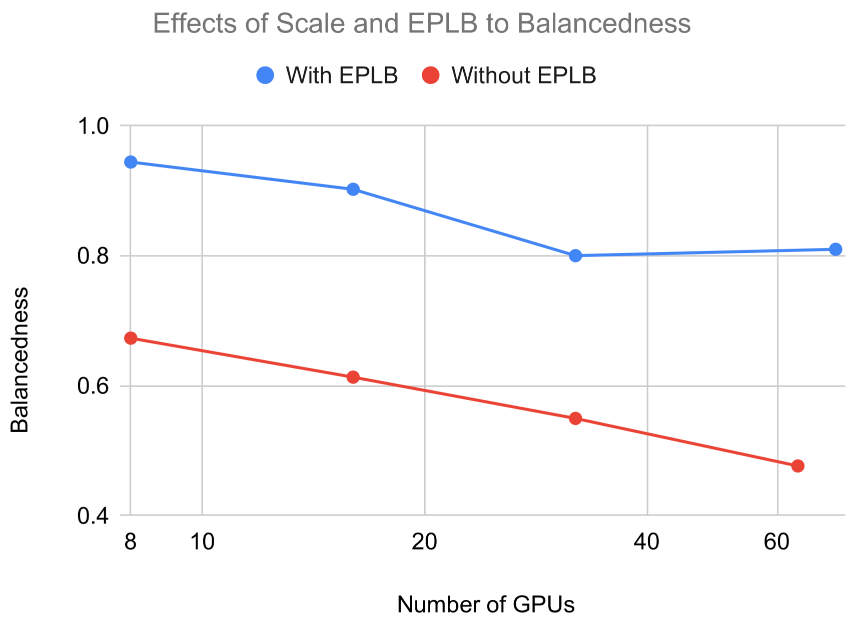


Figure 12: *Achieved expert balancedness given a fixed number of devices using expert parallel load balancing. Source [9]*

3.5 From Parts to the Whole

Given all of these considerations, we implemented a theoretical model estimating the model throughput given the hardware. It should make it easier to understand the tradeoffs between the latency, throughput, and cost between different hardware providers.

The model includes a number of assumptions, such as:

- All weights are stored in FP8; the MLA is computed in BF16; the matrix multiplications in the expert layers are performed in FP8. The communication is done in FP8 apart from the dispatch which is done in BF16.
- We make some strong assumptions about the inefficiency of FLOPs, memory bandwidth, and communication. We assume the same level of inefficiency across different hardware to make it fair. The levels are arbitrary and arguably can be one of the leading sources of error in our calculation. These inefficiencies in reality are also not the same for every hardware and are strongly dependent on the implementation.
- To make the calculations simpler we assume that no MTP is performed. We managed to make it run with MTP; however, we deemed the performance gains not worth the increase in complexity of the model, especially for larger batches.
- We only looked at the decode performance without taking into account prefilling.
- We assume no compute and memory loading overheads from the DeepEP two-batch communication library. This is not true, as these operations start a non-significant number of CUDA kernels, which can have downstream effects on highly optimized kernels like GEMM, as they will no longer get the expected number of threads.

At a high level, the performance model comprises three primary execution components: MLA computation, expert parallel (EP) computation, and communication overhead. For both MLA and EP operations, we determine whether memory bandwidth or computational throughput constitutes the limiting factor. Communication can be optionally overlapped using two-batch overlap (TBO), where total execution time for one forward pass becomes $T_{TBO} = 2 \times T(B/2)$.

The high-level model structure follows:

$$T^{MLA}_{bottleneck}(B,S) = \max\left(T^{MLA}_{mem}(B,S), T^{MLA}_{comp}(B,S)\right) \tag{46}$$

$$T^{EP}_{bottleneck}(B,S) = \max\left(T^{EP}_{mem}(B,S), T^{EP}_{comp}(B,S)\right) \tag{47}$$

$$T^{\text{no }TBO}_{total}(B,S) = T^{MLA}_{bottleneck}(B,S) + T^{EP}_{bottleneck}(B,S) + T_{comms}(B) \tag{48}$$

$$T^{TBO}_{total}(B,S) = 2 \times \max\left(T^{MLA}_{bottleneck}\left(\frac{B}{2}, S\right) + T^{EP}_{bottleneck}\left(\frac{B}{2}, S\right), T_{comms}\left(\frac{B}{2}\right)\right) \tag{49}$$

Computational, memory, and communication time estimates use the previously derived formulae, adjusted for real-world implementation inefficiencies. Practical systems rarely achieve theoretical peak performance, necessitating inefficiency factors across all components. The DeepEP communication library documentation indicates 40 GB/s achieved throughput from 50 GB/s theoretical peak, yielding a communication inefficiency factor of 1.25. FlashMLA achieves approximately 66% MFU, resulting in an MLA computation inefficiency factor of 1.5. Expert-layer computational performance, based on DeepGEMM benchmarks showing 1550 TFLOPs from 1980 TFLOPs theoretical FP8 dense peak performance results in an EP computation inefficiency factor of 1.3. Both computational inefficiencies receive an additional 10% penalty to account for suboptimal input conditions and overhead between the kernels. Memory inefficiency estimation proves more challenging without profiling. Due to some operations, such as matrix multiplication, requiring multiple loading operations for the same value, and the fact that most kernels are optimized for compute bound scenarios, we apply a conservative inefficiency factor of 2.0 to account for these overheads.

Token generation rates are calculated as the inverse of total execution time, with global throughput scaling by concurrent batch size:

$$TPS_{request} = \frac{1}{T_{total}^{TBO}} \tag{50}$$

$$TPS_{global} = \frac{1}{T_{total}^{TBO}} \times B \tag{51}$$

It is important to note that this model does not consider whether the proposed configurations are viable under real-world memory constraints. For instance, long context sequences can drastically reduce the maximum number of concurrent sequences due to memory limitations, resulting in significantly lower throughput than theoretical predictions.

3.6 Predictions and Real-World Comparison

To validate our theoretical model we compare it to real world measurements using three vastly different hardware setups:

- 4x8 H100: This is the basic setup that we consider reasonable to maintain by a large enterprise. This was also the setup we managed to obtain, therefore we have the measurements for all reasonable batch sizes.
- 9x8 H100: This is the setup from the SGLang blog post including their tuned performance measurements.
- 12x4 B200: This involves 48 out of 72 GPUs in an NVL72 setup. We use this to visualize how differently the new generation of hardware performs. This is also a setup tested by the SGLang team.

Figures 13 and 14 demonstrate that our theoretical model achieves reasonable agreement with empirical measurements. The first figure presents a systems total throughput and tokens per second (TPS) per request, while the second emphasizes efficiency by showing TPS per GPU.

As anticipated, our model overestimates actual performance by a considerable margin and we have to tune the model with our inefficiency factors. This discrepancy arises from two primary factors: first, individual component kernels fail to achieve peak performance as discussed previously; and second, peak performance of these individual components is rarely attained with constrained batch sizes. Furthermore, end-to-end optimization is often suboptimal, resulting in kernels optimized for different operational scenarios. These factors justify our incorporated inefficiency assumptions.

Small batch size estimation proved particularly challenging, as illustrated in Figure 13. At batch size 32, actual performance in our setup (shown in blue) exceeds theoretical predictions (given our inefficiency factors; it does not exceed the upper bound posed by the hardware itself). During modeling, we assumed uniform expert activation probability, which does not reflect reality. In practice, fewer experts are activated, resulting in higher throughput than predicted. As batch size increases, throughput converges to predicted levels, indicating activation of most to all available experts.

Consistent with our stated assumptions, the model does not assess whether given batch sizes are practically feasible under given systems memory constraints. In our system configuration, sequence eviction begins after batch size 1024, causing a sharp decline in per-request throughput and total throughput saturation. Increasing node count expands available KV-cache memory, enabling larger batch sizes as demonstrated by the two SGLang configurations.

4 Throughput: Theory vs Practice

Examining Figure 14, we observe that increasing batch size per GPU improves system efficiency substantially. However, realizing these optimal batch sizes necessitates extensive memory allocation for KV-cache storage. Given that total weight size remains largely static (excluding data-parallel MLA weights), distributing computation across additional GPUs reduces the per-GPU weight burden proportionally.

An increasing problem that large-scale systems pose is the communication overhead, which scales linearly with batch size. Consequently, configurations with large batch sizes and short sequence lengths may encounter communication bottlenecks. This phenomenon manifests in Figure 14, where the 4x8 H100 configuration achieves higher per-GPU throughput at batch size 512 compared to the 9x8 H100 setup, because the latter becomes communication-bound. Nevertheless, the former configuration cannot sustain these batch sizes in practice and will evict sequences, effectively running at smaller batch sizes. This also demonstrates the advantages of the NVL72 super node for inference workloads, effectively mitigating potential communication constraints.

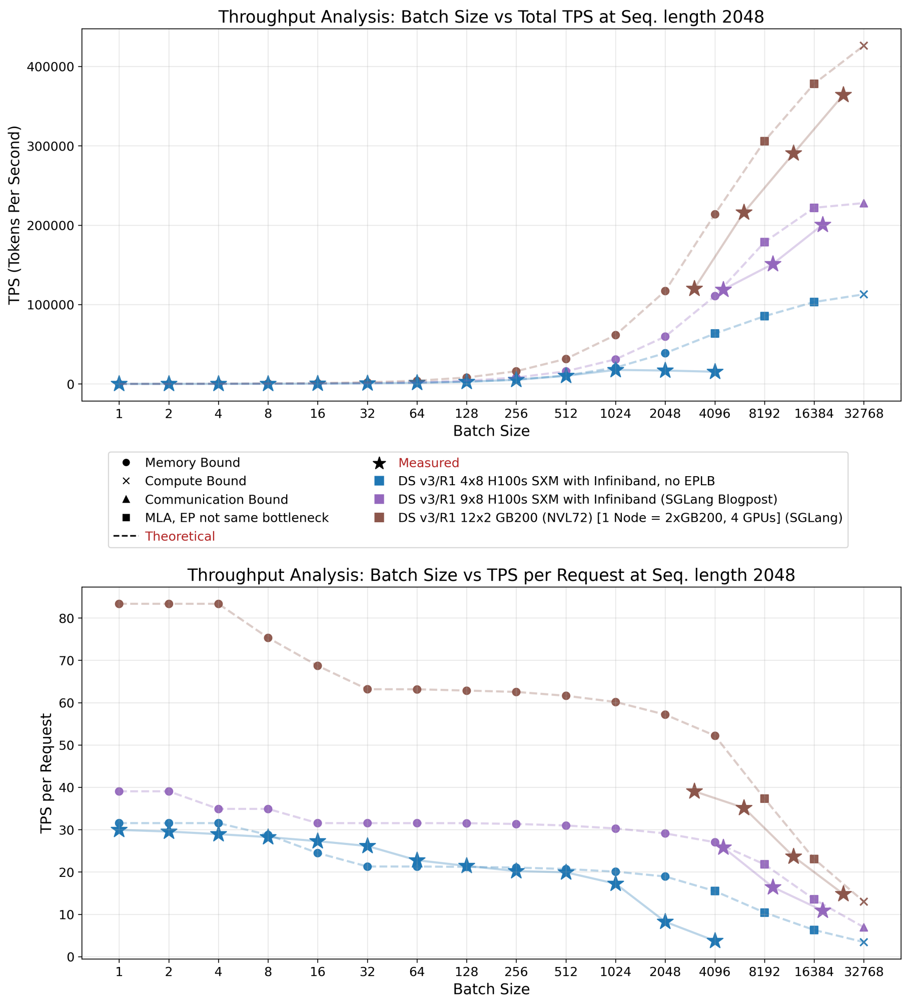


Figure 13: *Comparing the theoretical model with real world measurements for decode throughput performance. The measurements for the 4x8 H100 setup were conducted using our tokenomics benchmark, running AIME requests against the model. Data for the two other setups stems from the SGLang blog post [8, 9]. Our model works well in different scenarios on multiple hardware setups.*

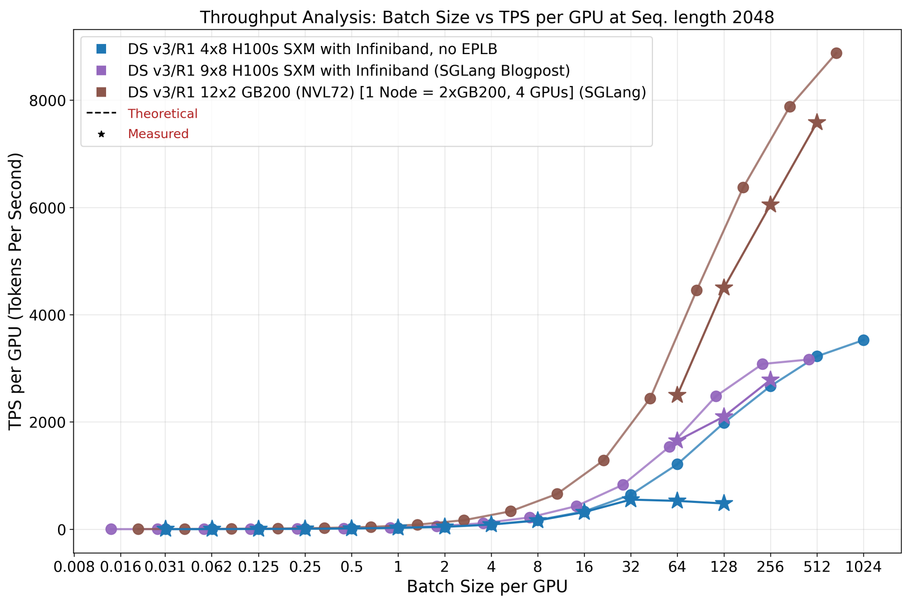


Figure 14: *Per-GPU efficiency across all configurations demonstrates consistent scaling characteristics. Measurements on our system (in blue) exhibit a performance ceiling due to token memory constraints that trigger sequence evictions, preventing running the full batch concurrently. This limitation clearly illustrates that additional memory capacity would enable our system to achieve higher throughput levels, validating the case for larger setups.*

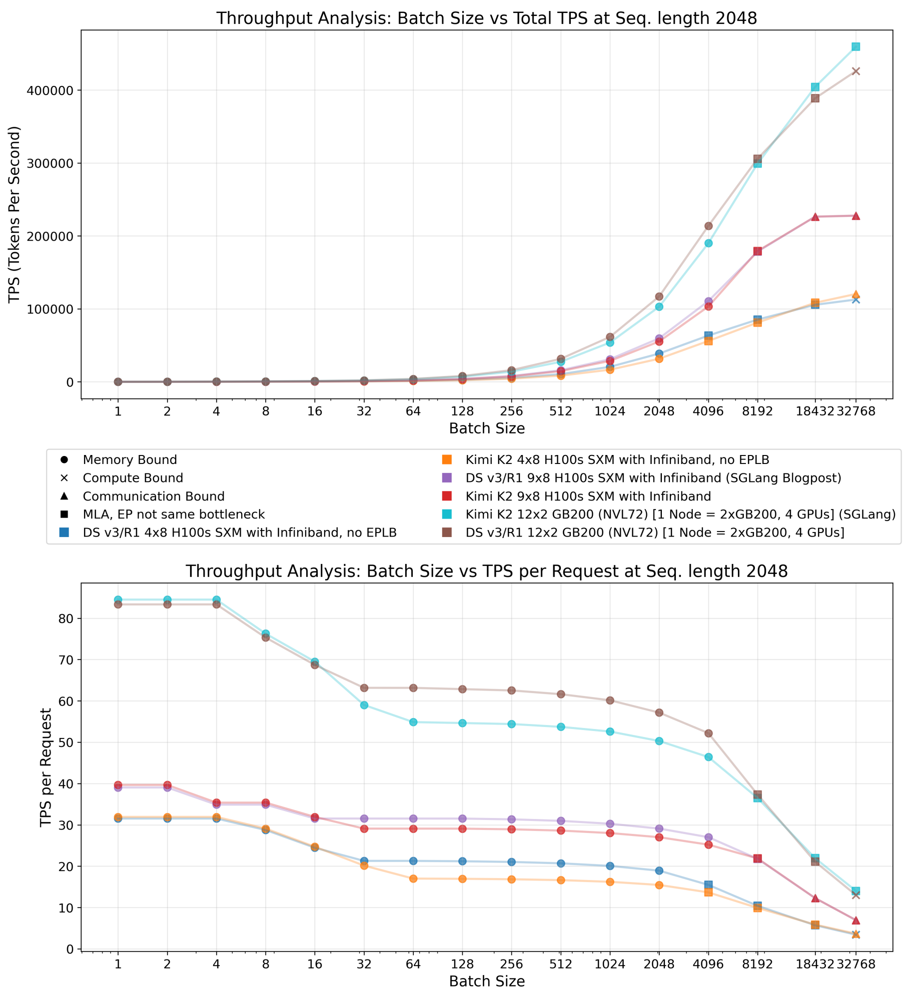


Figure 15: *Comparison of theoretical predictions for Kimi K2 to DeepSeek v3. Even though Kimi K2 has a lot more weights, it is only slightly larger due to the MLA and with the communication being the same in both models. However, we do not model if a given setup would actually have enough memory and thus would be able to run a given batch size. Kimi K2 would reach this limit much earlier leading to more evictions and smaller achievable batch sizes.*

Kimi-K2 [7] represents the first open-source LLM surpassing 1T parameters. The model employs the essentially identical architecture to DeepSeek v3, just with more routed experts per layer. As demonstrated in Figure 15, this configuration yields reduced throughput, particularly under memory-bound conditions where MLA runtime remains minimal. However, achieving equivalent batch sizes across identical hardware configurations is infeasible for large batches, as Kimi-K2 requires greater GPU memory allocation for storing its weights. Consequently, while theoretical performance degradation appears modest, practical performance disparities may be more pronounced due to reduced effective batch sizes compared to DeepSeek v3 deployment.

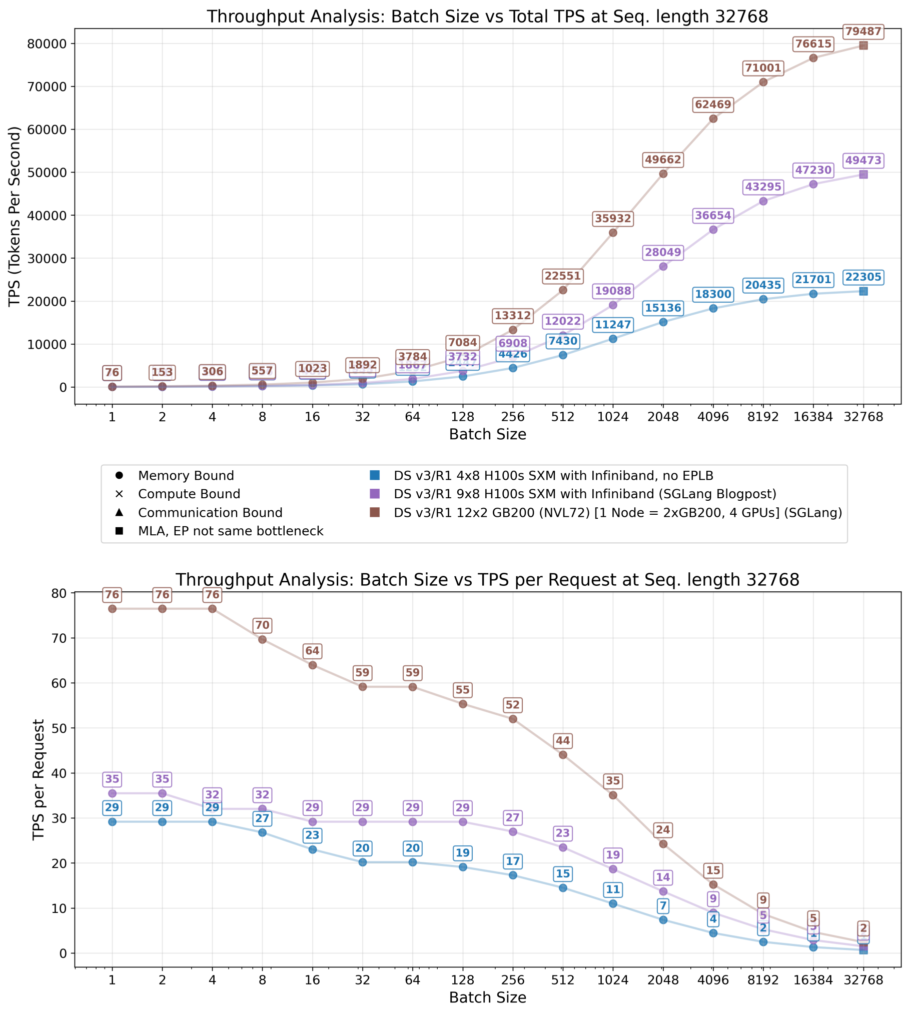


Figure 16: *Serving long contexts affects decode performance less than prefill, since the prefill runtime scales quadratically with input sequence length. However, a much bigger factor for decode, one we do not currently model, is the size of the KV cache. A large KV cache renders the MLA memory-bound for larger batch sizes and additionally forces running at smaller effective batch sizes, preventing the operation from becoming compute-bound in any realistic setup.*

Similar challenges with increased evictions and diminished effective batch sizes emerge when serving long sequences, as their KV cache demands substantial memory space. Although sequence length exerts a comparatively small impact on decode performance, as shown in Figure 16 (while exhibiting quadratic dependency during prefill on the sequence length), it constrains to running very low batch sizes, significantly reducing system efficiency. For example, a 4×8 H100 setup provides roughly 20 GB of GPU memory per GPU for the KV cache. At a context length of 32,768 tokens, this translates to a maximum effective batch size of $\frac{20\times10^9\times4\times8}{70\times10^3\times32768} \approx 279$. In practice, fragmentation of the KV cache and other inefficiencies reduce this number further. DeepSeek reports a much shorter average context length of 4989 tokens [2], which remains within manageable parameters.

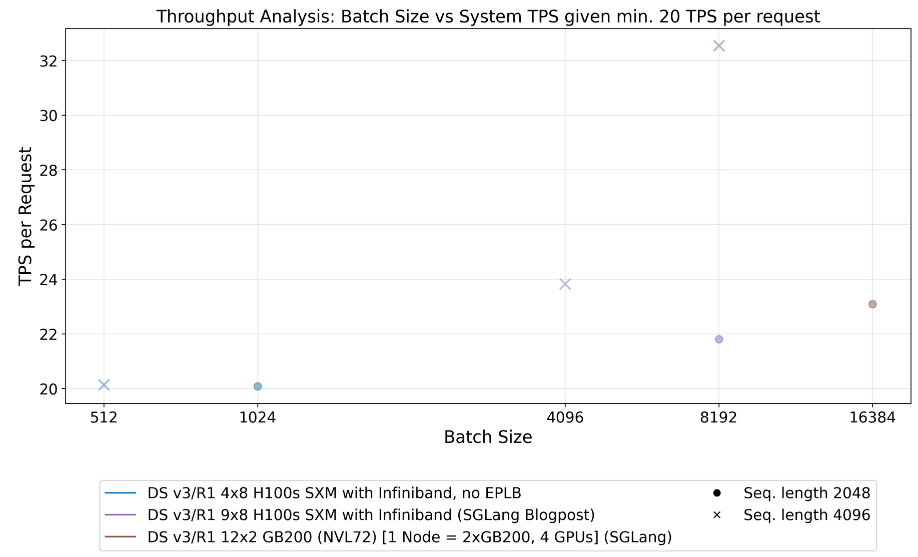


Figure 17: *The plot shows the maximum achievable batch size for running with at least 20 TPS per request. This limit is arbitrary but seems close to what providers often offer as an SLA. Again, the setup using the NVL72 outshines the competition, allowing for more efficient serving given an SLA.*

Production serving environments typically operate under SLA requirements that mandate minimum TPS thresholds per request. As illustrated in Figure 17, these performance guarantees often impose surprisingly restrictive limits on achievable batch sizes. Providers find themselves constrained to operate with smaller batches to meet per-request latency requirements, resulting in suboptimal efficiency. This constraint disproportionately affects smaller deployment configurations, creating a natural advantage for large-scale enterprise operations, serving to hundreds of thousands of customers.

Our previous analysis has given limited attention to the prefill phase. During prefill, the system computes the complete KV cache for all input tokens and generates the first output token. The computational complexity of this phase scales quadratically with sequence length due to the full attention computation required. For shorter sequences, prefill duration remains substantially shorter than the subsequent decode phase. However, in long-context scenarios, prefill can exceed decode time, creating significant system bottlenecks. Serving frameworks typically interrupt decode operations to process prefill batches, stalling the entire inference pipeline. Additionally, prefill operations are generally compute-bound rather than memory-bound, requiring distinct optimization strategies compared to decode. Large-scale deployments address this by implementing prefill-decode disaggregation, physically separating these phases across different instances. The prefill instance typically operates on fewer GPUs than the decode instance, reflecting the shorter duration and different resource requirements of prefill operations.

Interactive chat applications and agentic workflows frequently involve multi-turn sequences where consecutive requests share common prompt prefixes. Given the potential length of these conversational contexts, repeatedly executing prefill for shared content becomes highly inefficient. Sophisticated caching mechanisms can drastically improve performance by reusing computed KV caches across requests. Effective caching architectures extend beyond GPU memory to use CPU memory and even persisting to disk. Even disk-to-GPU transfers often outperform recomputation for sufficiently long sequences. Additionally on-disk caches can be held for longer, potentially for days. Such caching infrastructure can also serve as a buffer layer between disaggregated prefill and decode instances. Systems like LMCache and Mooncake provide foundational solutions to this problem. However, setting up such a caching infrastructure is non-trivial, and we save this topic for a future blog post. For the current analysis, we note that while prefill can substantially impact overall system performance, well-designed caching strategies offer substantial mitigation. DeepSeek's production deployment [2] reports achieving approximately 56.3% cache hit rates, demonstrating a good reduction in prefill time when deployed.

References

[1] CalvinXKY. *DeepSeek V3 MFU Calculator.* `https://calvinxky.github.io/mfu_calculation/deepseek3mfu.html`. 2025.

[2] DeepSeek-AI. *Day 6: One More Thing, DeepSeek-V3/R1 Inference System Overview.* `https://github.com/deepseek-ai/open-infra-index/blob/main/202502OpenSourceWeek/day_6_one_more_thing_deepseekV3R1_inference_system_overview.md`. Accessed: July 17, 2025. 2025.

[3] DeepSeek-AI. *DeepSeek Profile Data.* `https://github.com/deepseek-ai/profile-data`. Accessed: July 17, 2025. 2025.

[4] DeepSeek-AI et al. *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning.* 2025. arXiv: `2501.12948 [cs.CL]`. URL: `https://arxiv.org/abs/2501.12948`.

[5] DeepSeek-AI et al. *DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model.* 2024. arXiv: `2405.04434 [cs.CL]`. URL: `https://arxiv.org/abs/2405.04434`.

[6] DeepSeek-AI et al. *DeepSeek-V3 Technical Report.* 2025. arXiv: `2412.19437 [cs.CL]`. URL: `https://arxiv.org/abs/2412.19437`.

[7] Moonshot AI. *Kimi K2: Open Agentic Intelligence.* `https://moonshotai.github.io/Kimi-K2/`. Accessed: July 21, 2025. 2025.

[8] SGLang. *Deploying DeepSeek on GB200 NVL72 with PD and Large Scale EP (Part I): 2.7x Higher Decoding Throughput.* `https://lmsys.org/blog/2025-06-16-gb200-part-1/`. Accessed: July 20, 2025. 2025.

[9] SGLang. *Deploying DeepSeek with PD Disaggregation and Large-Scale Expert Parallelism on 96 H100 GPUs.* `https://lmsys.org/blog/2025-05-05-large-scale-ep/`. Accessed: July 20, 2025. 2025.

[10] C. Zhao et al. *DeepEP: an efficient expert-parallel communication library.* `https://github.com/deepseek-ai/DeepEP`. 2025.

[11] C. Zhao et al. "Insights into deepseek-v3: Scaling challenges and reflections on hardware for ai architectures". In: *Proceedings of the 52nd Annual International Symposium on Computer Architecture.* 2025, pp. 1731–1745.