September, 2025

InferSim: A Lightweight LLM Inference Performance Simulator

Alimama AI Infra Team[*] and **Future Living Lab**[*]

[*]Alibaba Group

Abstract: The inference and serving costs of large language models (LLMs) are extremely high, especially as modern LLMs continue to grow in size and context length. Quantitative analysis of LLM inference and the model-system co-design informed by it are therefore essential. We introduce *InferSim*, a lightweight LLM inference performance simulator implemented in pure Python with zero third-party dependencies, to predict the inference performance of LLM models rapidly and accurately. Experiments show that the performance predicted by the simulator closely aligns with the actual performance (the simulated throughput differs by only **15%** from DeepSeek-V3's profile data, and **4%** to **8%** from the measured performance of Qwen3-30B-A3B and Qwen3-8B on SGLang).

Code is available at https://github.com/alibaba/InferSim.

1 Introduction

InferSim is a lightweight simulator for LLM inference, written in pure Python without any third-party depenencies. It calculates TTFT (Time To First Token), TPOT (Time Per Output Token) and throughput based on computational complexity FLOPs (Floating-Point Operations), GPU computing power FLOPS (Floating-Point Operations per Second), GPU memory bandwidth, and MFU (Model FLOPs Utilization) obtained by benchmarking the state-of-the-art LLM kernels. For multi-GPU, multi-node deployment, *InferSim* also estimates the communication latency according to data volume and bandwidth.

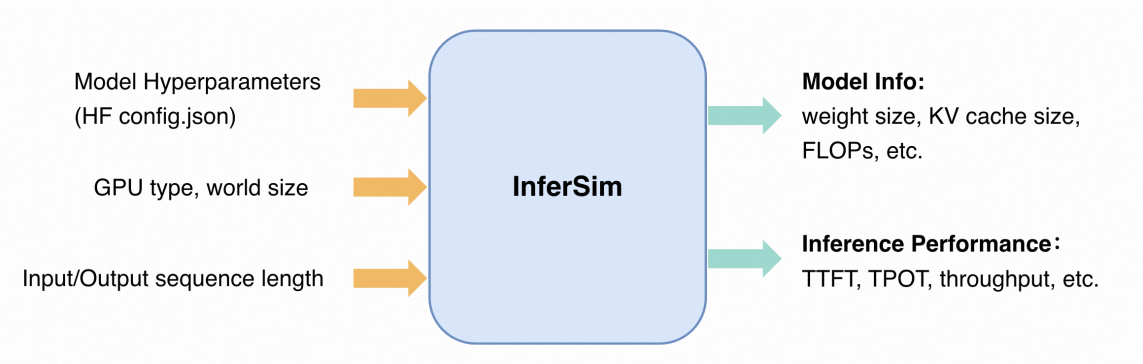

Figure 1: Introduction of *InferSim*

The main use cases of *InferSim* include:

- **Model-system co-design**: predicting inference performance given the hyperparameters of a model.
- **Inference performance analysis**: quantifying performance bottlenecks, such as compute-bound or IO-bound, and supporting optimization efforts.

2 Methodology

The basic components of LLM include attention, FFN (Feed Forward Network)/MoE (Mixture of Experts), and also communication for multi-GPU or multi-node inference.

Attention. The latency of a multi-head attention can be calculated according to computational complexity FLOPs, GPU computation power FLOPS and MFU. As

for the computing bottleneck and the memory access bottleneck, they are both reflected in the MFU. Usually, the MFU is high under the computing bottleneck and low under the memory access bottleneck.

$$FLOPs_{attn\_core} = 4n_h S_q S_{kv} d_h R_{mask}, \quad t_{attn\_core} = \frac{FLOPs_{attn\_core}}{FLOPS_{gpu} \times MFU_{attn\_core}}$$

$$FLOPs_{qkvo\_proj} = 4(n_h + n_{kv}) d_h d_{hidden}, \quad t_{attn\_proj} = \frac{FLOPs_{qkvo\_proj}}{FLOPS_{gpu} \times MFU_{attn\_proj}}$$

$$t_{attn} = t_{attn\_core} + t_{attn\_proj}$$

where S_q, S_{kv} is the sequence length of Q and KV respectively, d_h is the head dimension, n_h is the number of attention heads, n_{kv} is the number of KV heads, d_{hidden} is the hidden size, R_{mask} is the causal mask ratio, which is normally 0.5 for prefilling and 1 for decoding. $attn\_core$ is the $softmax(QK^T)V$, $qkvo\_proj$ represents the linear projections of Q, K, V and O.

MoE/FFN. MoE is more widely used in recent LLM models like DeepSeek-V3[2] and Qwen3[14]. Dense FFN can be treated as a special MoE with only one expert.

$$FLOPs_{MoE} = 6S_q d_{hidden} d_{inter} n_{act}$$

$$t_{MoE} = \frac{FLOPs_{MoE}}{FLOPS_{gpu} \times MFU_{MoE}}$$

where d_{inter} is the intermediate dimension of each expert, n_{act} is the number of activated experts per token.

Communication. In EP (Expert Parallelism) of MoE model, the communication operations before and after the MoE layer correspond to dispatch and combine, respectively. We can estimate the communication time based on the data volume and

bandwidth. Intra-node communication utilizes NVLink, while inter-node communication employs RDMA, each offering distinct bandwidth capabilities.

$$t_{comm} = \frac{N_{bytes}}{Bandwidth}$$

With the latency of the basic modules, we can estimate the end-to-end time consumption and throughput. The performance of prefilling and decoding should be estimated separately, as they have different computational characteristics[6].

2.1 Prefilling

For an LLM model with L layers of transformers, the TTFT and throughput are as follows:

$$TTFT = (t_{attn} + t_{MoE} + t_{comm}) \times L + t_{schedule}$$

$$Throughput = \frac{S_q}{TTFT}$$

where $t_{schedule}$ is the scheduling overhead, and it usually takes several milliseconds to tens of milliseconds, S_q is the number of input tokens for one prefilling.

2.2 Decoding

For decoding stage, the formula is similar to that of prefilling, however, the MFUs in each module are different, and the $t_{schedule}$ is usually smaller.

$$TPOT = (t_{attn} + t_{MoE} + t_{comm}) \times L + t_{schedule}$$

$$Throughput = \frac{B}{TPOT}$$

where B is the decoding batch size. B is constrained by the size of KV cache and GPU memory capacity.

2.3 Kernel Benchmark

The accuracy of the MFU directly determines the accuracy of the simulator results. Therefore, we benchmark the state-of-the-art kernel libraries to collect reliable MFU data. For attention, we benchmark MHA, GQA[1] as well as MLA[8] on FlashInfer[16] and FlashAttention-3[7]. For MoE GroupedGEMM and general GEMM kernels, we benchmark on DeepGEMM[9].

We have open-sourced the kernel benchmark scripts (Listed in Table 1) so that users can collect MFU data under different hyperparameters.

Kernel	Stage	Library
MHA/GQA	Prefill/Decode	FlashInfer, FlashAttention-3
MLA	Prefill/Decode	FlashInfer, FlashAttention-3
GroupedGEMM	Prefill/Decode	DeepGEMM
GEMM	Prefill/Decode	DeepGEMM

Table 1: A list of the kernel benchmark scripts.

Based on the roofline model and benchmark results, we analyze the performance of GQA, MLA, and MoE on H20 and H800 GPU.

GQA. For GQA in the decoding phase, in order to achieve a balance between computation and memory access, the relationship between group size g and computation bandwidth ratio R is as follows:

$$\frac{4n_h d_h S}{2n_{kv} d_h S} = \frac{FLOPS}{Bandwidth} = R \Rightarrow g = \frac{d_h}{d_{kv}} = \frac{R}{2}$$

where n_h and n_{kv} is the number of Q and KV heads respectively, d_h is head dimension, S is the sequence length. The roofline (computation-bandwidth ratio R) is 74 for H20 and 591 for H800. Hence theoretically, H20 achieves an optimal computation-bandwidth balance at a group size of 37, whereas the H800's balance point lies at a group size of 296.

Figure 2 demonstrates the GQA's MFU under varying group sizes in the decoding stage, n_{kv} is fixed at 2, n_h ranges from 16 to 256, resulting in group sizes spanning from 8 to 128. It can be observed that the MFU of H20 stops growing after reaching

$g = 64$, while the MFU of H800 keeps rising within $g = 128$, which aligns with the prior theoretical analysis.

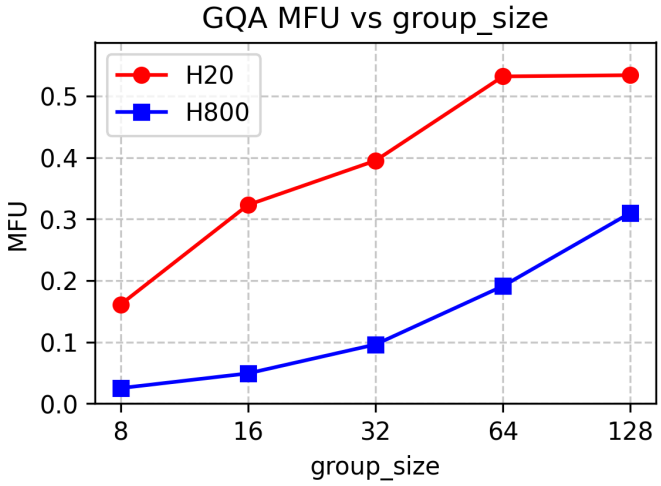

Figure 2: GQA's MFU on different group sizes in the decoding stage, where $d_h = 128$, $n_{kv} = 2$, sequence length is 4096 and batch size is 256. MFU data is obtained by benchmarking FlashInfer and XQA[5].

MLA. MLA employs distinct computational paradigms between its prefilling and decoding stages. During prefilling, the low-rank KV matrix undergoes up-projection to $n_h \times d_h$ for MHA computation. In decoding, the up-projection parameters are absorbed into the Q and O projection weights, enabling efficient MQA computation directly on the original low-rank KV, therefore, in order to balance computation and memory access, the relationship between computation-bandwidth ratio R and n_h should be as follows:

$$\frac{2n_h(2d_c + d_r)S}{(d_c + d_r)S} = \frac{FLOPS}{Bandwidth} = R \Rightarrow n_h \approx \frac{R}{4}$$

where d_c is the low-rank dimension of KV, d_r is the decoupled RoPE dimension of K. Since d_r is much smaller than d_c, n_h is approximately equal to $\frac{R}{4}$. Consequently, the computation-bandwidth balanced n_h is 19 for H20 and 148 for H800.

Figure 3 shows MLA's MFU on different number of heads (n_h ranges from 16 to 128). On H20, a relatively high MFU is achieved when $n_h = 16$. While on H800,

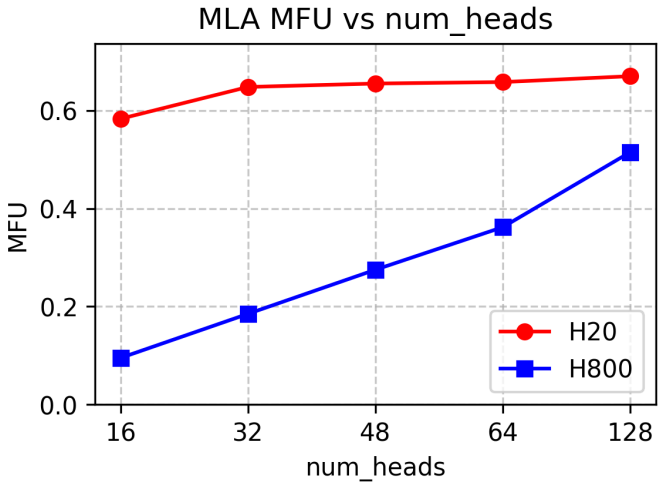

Figure 3: MLA's MFU on different number of heads in the decoding stage, where $d_c = 512$, $d_r = 64$, sequence length is 4096 and batch size is 256. MFU data is obtained by benchmarking FlashInfer and FlashMLA-ETAP[3].

MFU reaches 0.5 when $n_h = 128$. The results correspond well to the theoretically derived balance points, 19 on H20 and 148 on H800.

MoE. Modern LLMs tend to use highly sparse MoE architectures. However, when designing sparsity, the computing power and bandwidth of the hardware must also be considered. We define the sparsity of MoE as S. The relation between S, computation-bandwidth ratio R and total decoding batch size B_{MoE} is as follows:

$$S = \frac{n_{act}}{n_e}$$

$$\frac{2B_{MoE}d_{hidden}d_{inter}n_{act}}{d_{hidden}d_{inter}n_e} = \frac{FLOPS}{Bandwidth} = R \Rightarrow B_{MoE} = \frac{R}{2S}$$

where n_{act} is the number of experts activated per token, n_e is the total number of experts. Taking DeepSeek-V3 as an example, with $S = 8/256$, deployed on H800, the B_{MoE} needs to reach 9456. Assuming the batch size per GPU is 128, then at least EP74 is required. If H20 is used instead, B_{MoE} can be reduced to 1184, and the required EP size is 19 when the batch size per GPU is 64.

Figure 4 shows the MoE Grouped GEMM's MFU on different EP sizes during de-

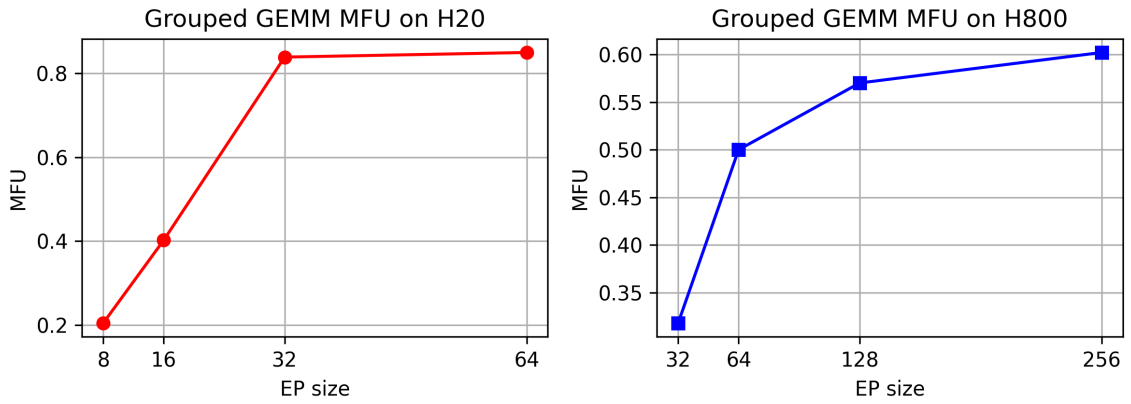

Figure 4: MoE GroupedGEMM's MFU on different EP sizes during decoding, where $S = 8/256$, batch size per GPU is 64 for H20 and 128 for H800. MFU data is obtained by benchmarking DeepGEMM.

coding phase. Relatively high MFU is achieved at EP32 on H20 and EP128 on H800, where the total batch size is 2048 and 16384 respectively. These results validate our theoretical analysis of the minimum total batch size 1184 on H20 and 9456 on H800.

3 Use Cases

There are two main use cases of *InferSim*, model-system co-design and inference performance analysis.

3.1 Model-system Co-design

Figure 5 depicts a simple example of model-system co-design using *InferSim*. Model designer provides an initial version of model hyperparameters, determines the GPU type and quantity to deploy, as well as input/output sequence length, target throughput, TPOT, TTFT, etc. according to business scenarios. Given these inputs, *InferSim* can predict the inference performance metrics, including throughput, TPOT, TTFT, etc. If the predicted performance does not meet the requirements, the model designer needs to iteratively fine-tune the hyperparameters until the targets are satisfied.

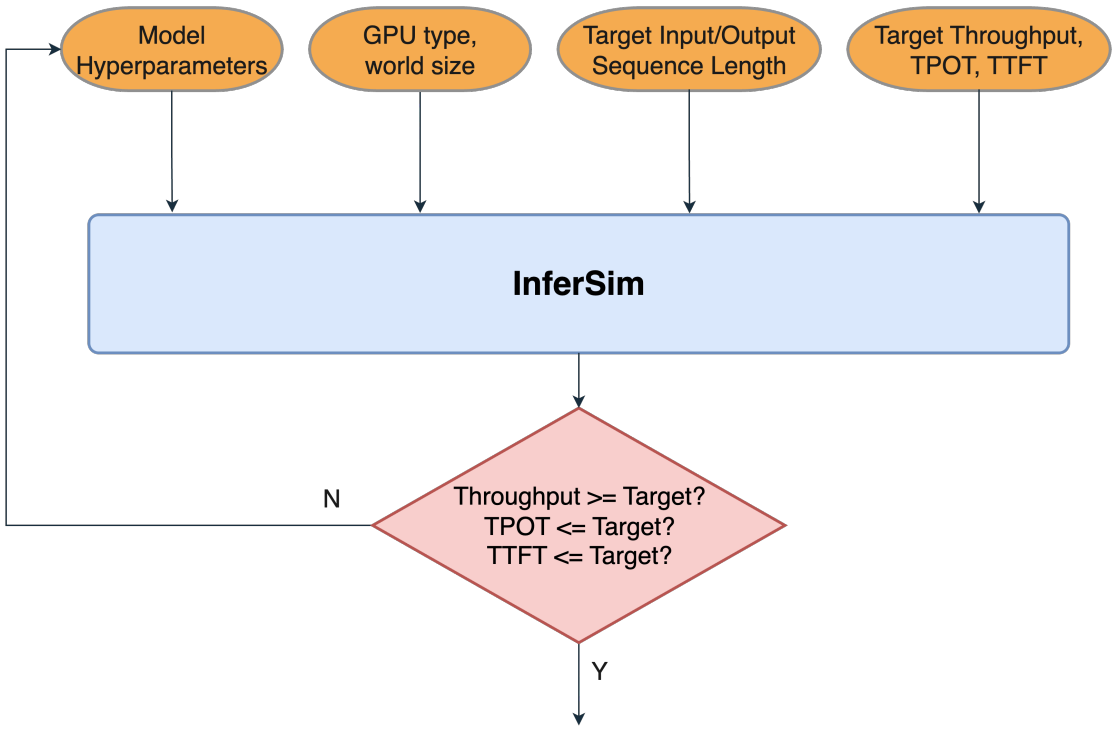

Figure 5: An example of model-system co-design using *InferSim*.

3.2 Inference Performance Analysis

InferSim can predict the inference latency for each module, including attention, FFN/MoE and communication, etc. Users can compare the actual inference timeline with the simulation results to identify potential performance optimization points.

More deeply, to analyze the compute-bound or IO-bound scenarios, we can compare the arithmetic intensity (the ratio of arithmetic operations to the number of bytes accessed from memory) with the hardware's computation-bandwidth ratio. For specific operations, such as attention or MoE grouped GEMM, *InferSim* can calculate the $FLOPs$ and memory access size N_{bytes}, and compare them with the computation-bandwidth ratio of the hardware.

$$x = \frac{FLOPs}{N_{bytes}} - \frac{FLOPS}{Bandwidth}, \begin{cases} \text{compute bound} & x > 0 \\ \text{balance} & x = 0 \\ \text{IO bound} & x < 0 \end{cases}$$

When the arithmetic intensity exceeds the computation-bandwidth ratio, the operation becomes compute-bound; conversely, when the arithmetic intensity is lower than the computation-bandwidth ratio, it becomes IO-bound. Optimal hardware performance is achieved when these two values are closely balanced.

4 Evaluation

We evaluate the simulator result on DeepSeek-V3 and Qwen3-30B-A3B respectively. It shows that our simulator results are close to the actual data. The simulated prefilling and decoding TGS (Tokens/GPU/Second) of DeepSeek-V3 differ by only **15%** from DeepSeek-V3's profile data[10]. While the simulated TGS of Qwen3-30B-A3B and Qwen3-8B differ by only **4%** to **8%** from the actual data tested with SGLang[12].

Model	**GPU**	**Prefill TGS (Actual)**	**Prefill TGS (Sim)**	**Decode TGS (Actual)**	**Decode TGS (Sim)**
DeepSeek-V3	H800	7839	9034	2324	2675
Qwen3-30B-A3B	H20	16594	17350	2749	2632
Qwen3-8B	H20	15061	16328	2682	2581

Table 2: The comparison of simulator result with actual data.

For DeepSeek-V3, we use the same deployment configuration as DeepSeek-V3's profile data[10]. For prefilling, the configuration uses EP32 and TP1 with a 4K prompt length and 16K tokens per GPU. For decoding, it uses EP128 and TP1 with a 4K prompt length and 128 requests per GPU. Two micro-batches are used to overlap computation and all-to-all communication in both prefilling and decoding stage. However, the all-to-all implementations in DeepEP[2] for prefilling and decoding are different, the communication during decoding does not occupy GPU SMs, while prefilling occupies 24 SMs, this difference is also considered in our *InferSim*.

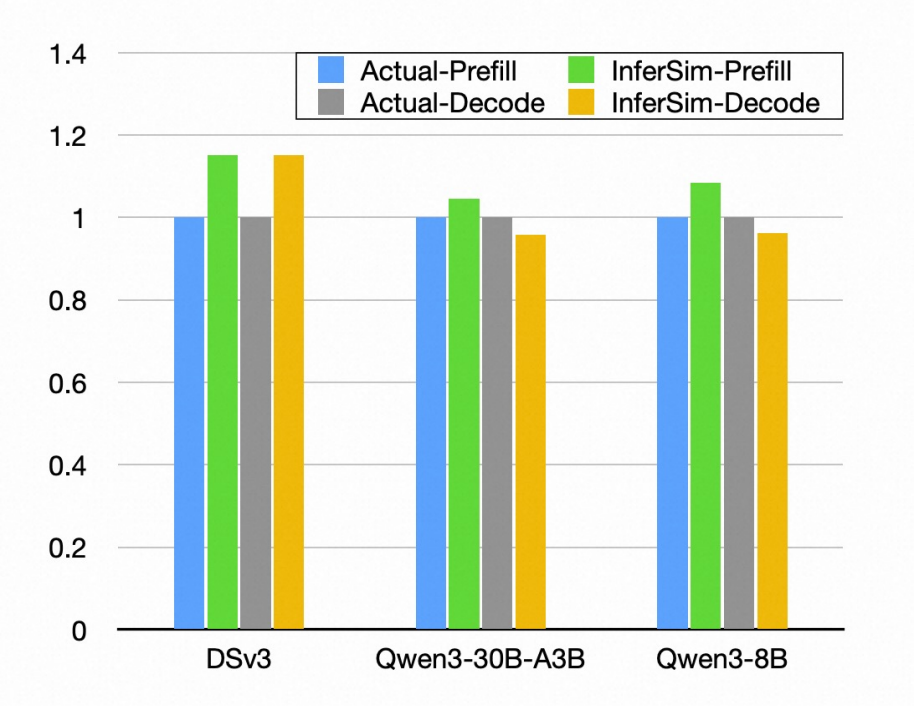

Figure 6: The comparison of simulator result with actual data (normalized by actual data).

For Qwen3-30B-A3B (BF16), we test the actual throughput with SGLang and simulate with same setup. The prefilling stage is configured with TP1, a 4K prompt length, and processes 16K tokens per GPU. In contrast, the decoding stage uses DP4 and EP4, with 4K prompt length and 2K output length, and handles 100 requests per GPU.

For Qwen3-8B (FP8 GEMM), most of the configurations are same as Qwen3-30B-A3B, the only difference is that Qwen3-8B uses TP1 for both prefilling and decoding.

5 Conclusion

In this report, we introduce *InferSim*, a lightweight LLM inference simulator written in pure Python. It is very easy to use, yet the simulation results are accurate, since the MFU data is collected by benchmarking and analyzing the state-of-the-art kernels. Experiments show that the simulated throughput of DeepSeek-V3 differs by

only **15%** from DeepSeek-V3 profile data[10], meanwhile that of Qwen3-30B-A3B and Qwen3-8B differ by only **4%** to **8%** from the actual data tested using SGLang.

Future work mainly includes the following aspects:

- Constructing more complete kernel benchmarks, to make MFU data cover more model hyperparameters and hardware.
- Supporting emerging model structures and algorithms, such as GDN[15] in Qwen3-Next, and MoE++[4] in LongCat-Flash[11].
- More complex collective communication and deployment methods, including TP, DP, PP, EP, and their combinations, as well as distributed deployment methods such as AFD (Attention-FFN Disaggregation)[13].

References

[1] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints, 2023.

[2] DeepSeek-AI. Deepseek-v3 technical report, 2024.

[3] Pengcuo Dege and Chang Kong. Flashmla-etap: Efficient transpose attention pipeline for accelerating mla inference on nvidia h20 gpus. `https://github.com/pengcuo/FlashMLA-ETAP`, 2025.

[4] Peng Jin, Bo Zhu, Li Yuan, and Shuicheng Yan. Moe++: Accelerating mixture-of-experts methods with zero-computation experts, 2024.

[5] NVIDIA. Xqa kernel. `https://github.com/NVIDIA/TensorRT-LLM/tree/main/cpp/kernels/xqa`, 2024.

[6] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: A kvcache-centric disaggregated architecture for llm serving. 2024.

[7] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony

and low-precision, 2024.

[8] DeepSeek Team. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model, 2024.

[9] DeepSeek Team. Deepgemm: clean and efficient fp8 gemm kernels with fine-grained scaling. In *https://github.com/deepseek-ai/DeepGEMM*, 2025.

[10] DeepSeek Team. Profiling data in deepseek infra. In *https://github.com/deepseek-ai/profile-data*, 2025.

[11] Meituan LongCat Team. Longcat-flash technical report, 2025.

[12] SGLang Team. Sglang: a fast serving framework for large language models and vision language models. In *https://github.com/sgl-project/sglang*, 2025.

[13] StepFun Team. Step-3 is large yet affordable: Model-system co-design for cost-effective decoding, 2025.

[14] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025.

[15] Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated delta networks: Improving mamba2 with delta rule, 2025.

[16] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. Flashinfer: Efficient and customizable attention engine for llm inference serving. *arXiv preprint arXiv:2501.01005*, 2025.